An Introduction to Ada 95 for Programmers

(Part 1 – Basic Concepts and Intro to the Language)

Dr. David A. Cook
DCook@AEGISTG.COM

Dr. Eugene W.P. Bingue
Dr.Bingue@ix.netcom.com
Ada-Engineering.com

Ada
The Language For a Complex World

Introduction to Ada 95 for Programmers

- **Learning Objectives**
  - Upon completion of this course, participants will be able to:
    - Exploiting typing to improve safety and reliability
    - Develop programs exploiting features of Ada
    - Build modular programs using package units
    - Object-Oriented Programming
    - Application of annexes.

Review Software Engineering 101

- Software Engineering Goals
- Principles of Software Engineering
- Five Principles to Ensure proper Modularity
- Ariane 5 Explodes on First Flight

Software Engineering Goals

- ✓ Modifiability
- ✓ Efficiency
- ✓ Reliability
- ✓ Understandability

Principles of Software Engineering

- Abstraction
- Information Hiding
- Modularity
- Localization
- Uniformity
- Completeness
- Confirmability

Abstraction:

is to extract essential properties while omitting inessential details.

Information Hiding:

is to make inaccessible certain details that should not affect other parts of a system.

Modularity:

deals with how the structure of an object can make the attainment of some purpose easier.

Localization:

is primarily concerned with physical proximity.

Uniformity:

simply means that the modules use a consistent notation and are free from any unnecessary differences.

Completeness:

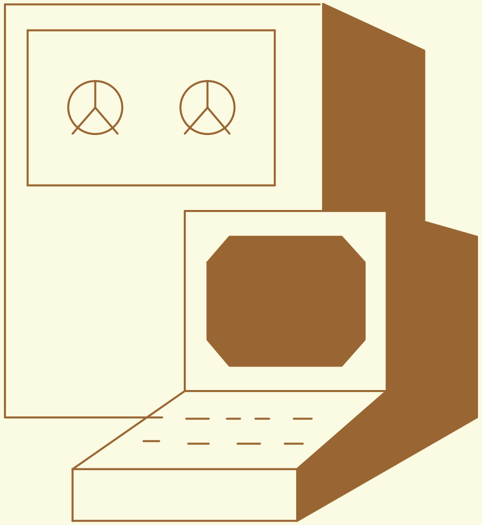
ensures that all of the important elements are present.

Confirmability:

implies that we must decompose our system so that it can be readily tested.

Five Principles to Ensure
Proper Modularity

- linguistic modular units (notation)
- few interfaces
- small interfaces (low coupling)
- explict interfaces
- information hiding



[Meyer88]

Linguistic Modular Units

The principle of linguistic modular units expresses that the formalism used to express designs, programs etc. must support the view of modularity retained.

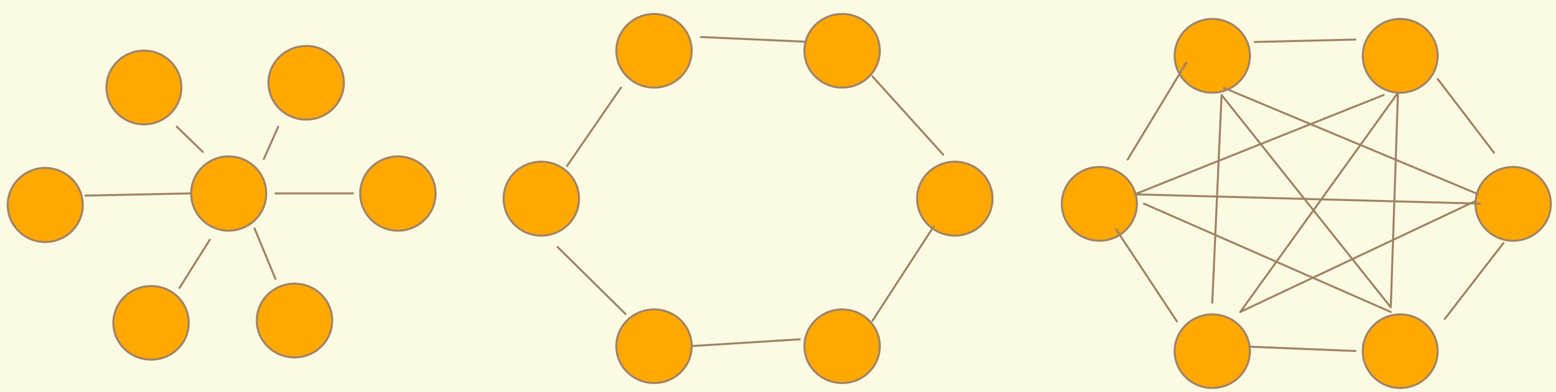
Modules must correspond to syntactic units in the language used

In the case of programming languages, modules should be separately compilable.

[Meyer88]

Few Interfaces

The "few interfaces" principle restricts the overall number of communication channels between modules in a software architecture.



min: (n-1) one boss

robust

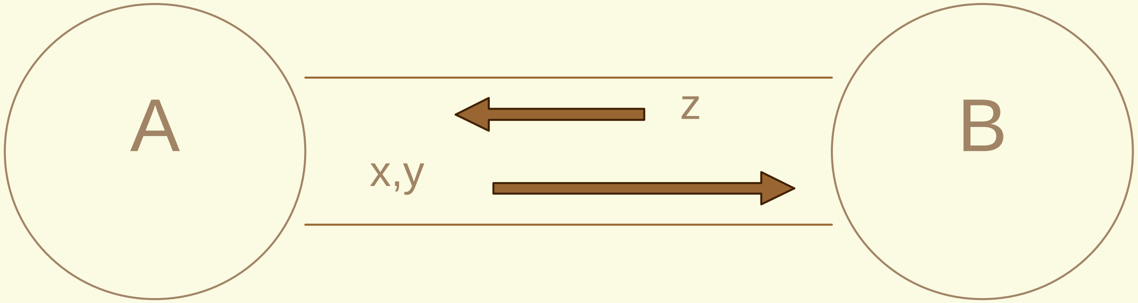
max: n(n-1)/2

Every module should communicate with as few others as possible

[Meyer88]

Small Interface (Loss Coulping)

The "Small Interface" or "Loss Coulping" principle relates to the size of intermodule connection rather than to their number.

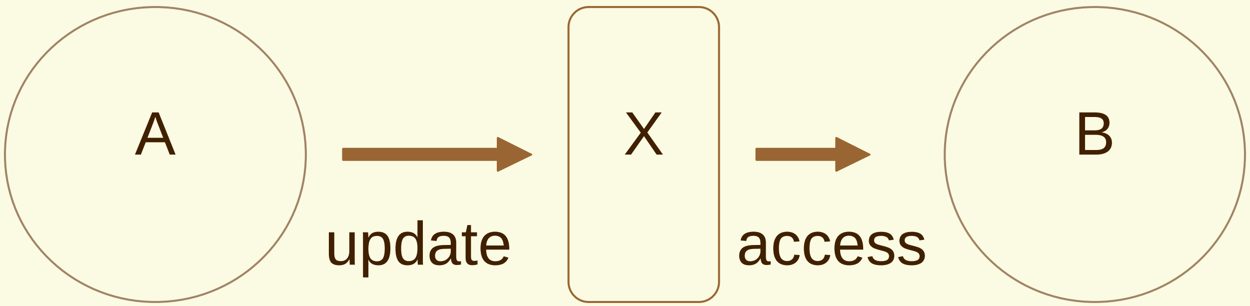


If any two modules communicate at all, they should exchange as little information as possible.

[Meyer88]

Explicit Interfaces

Whenever two modules A and B communicate, this must be obvious from the text of A or B or both.

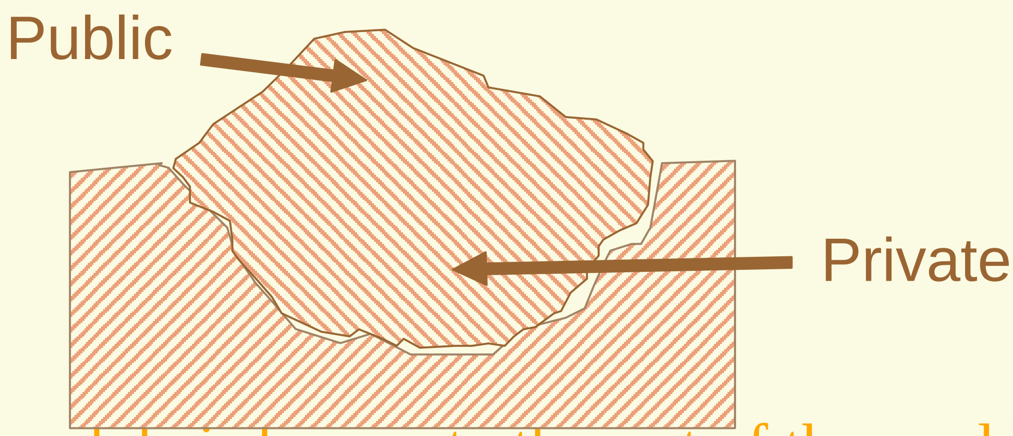


Data Sharing

[Meyer88]

Information Hiding

All information about a module should be private to the module unless it is specifically declared public.



Every module is known to the rest of the world (that is to say, to designers of other modules) through some offical description or interface.

[Meyer88]

Ariane 5 Explodes on First Flight

- Follow-on to Ariane 4
- Rocket and cargo worth $500 million
- Uninsured

Ariane 5

- Nominal behavior of the launcher up to H0 + 36 seconds
- Failure of the back-up inertial reference system followed immediately by failure of the active inertial reference system
- Swiveling into the extreme position of the nozzles of the two solid boosters and, slightly later, of the Vulcain engine, causing the launcher to veer abruptly

Ariane 5

✓ When the links between the solid boosters and the core stage ruptured, self-destruction of the launcher was correctly triggered automatically

Software Upgrade

- ✓ While a large amount of money had been spent to upgrade rocket hardware, software was another issue
- ✓ The view was that, unless proven necessary, it was not wise to make changes in software which worked well on Ariane 4
- ✓ The impact on the software of moving from the Ariane 4 to the Ariane 5 was thought to be so low that the expected trajectory data for the Ariane 5 was not even included in software requirements

Cause

- ✓ Nozzle deflections were commanded by the On-Board Computer (OBC) software on the basis of data transmitted by the active Inertial Reference System (SRI 2).
  - Part of these data showed a diagnostic bit pattern from the SRI 2 computer, which was interpreted as flight data.
- ✓ SRI 2 unit had declared a failure due to a software exception.
- ✓ The OBC could not switch to the back-up SRI 1 because that unit had already ceased to function during the previous data cycle (72 milliseconds period) for the same reason as SRI 2.
- ✓ The internal SRI software exception was caused during
  - Execution of a data conversion from 64-bit floating point to 16-bit signed integer value greater than what could be represented by a 16-bit signed integer.
  - This resulted in an Operand Error.
  - The data conversion instructions (in Ada code) were not protected from causing an Operand Error, although other conversions of comparable variables in the same place in the code were protected.

Cause

- The Operand Error occurred due to an unexpected high value of an internal alignment function result called BH, Horizontal Bias, related to the horizontal velocity sensed by the platform. This value is calculated as an indicator for alignment precision over time
- The value of BH was much higher than expected because the early part of the trajectory of Ariane 5 differs from that of Ariane 4 and results in considerably higher horizontal velocity values

Obsolete Requirement

- ✓ The error occurred in a part of the software that only performs alignment of the strap-down inertial platform. This software module computes meaningful results only before lift-off. As soon as the launcher lifts off, this function serves no purpose
- ✓ The original requirement for the continued operation of the alignment software after lift-off was brought forward more than 10 years ago for the earlier models of Ariane, in order to cope with the rather unlikely event of a hold in the count-down e.g. between - 9 seconds, when flight mode starts in the SRI of Ariane 4, and - 5 seconds when certain events are initiated in the launcher which take several hours to reset
- ✓ The period, 50 seconds after start of flight mode, was based on the time needed for the ground equipment to resume full control

Design Decisions

- ✓ In part, not all the conversions were protected because a maximum workload target of 80% had been set for the SRI computer
- ✓ In particular, the conversion of floating point values to integers was analyzed for seven variables for risk of an Operand Error
  - Protection was added to four of the variables
  - However, three of the variables were left unprotected
  - No reference to justification of this decision was found directly in the source code. Given the large amount of documentation associated with any industrial application, the assumption, although agreed, was essentially obscured, though not deliberately, from any external review. But all partners at several levels agreed to it
  - There is no evidence that any trajectory data were used to analyze the behavior of the unprotected variables

Lessons

- What lessons might you draw so far?

Specifications

- ✓ The partners jointly agreed not to include the Ariane 5 trajectory data in the SRI requirements and specification
- ✓ The systems specification of the SRI does not indicate operational restrictions that emerge from the chosen implementation

Design Philosophy

- In the event of any kind of exception, the system specification stated that: the failure should be indicated on the databus, the failure context should be stored in an EEPROM memory (which was recovered and read out for Ariane 501), and finally, the SRI processor should be shut down
- The reason behind this drastic action lies in the culture within the Ariane effort of only addressing random hardware failures. From this point of view exception - or error - handling mechanisms are designed for a random hardware failure which can quite rationally be handled by a backup system

Testing

- Numerous tests and simulations carried out at the Functional Simulation Facility (ISF)
- These included nominal, degraded, and failure scenarios
- In these tests many equipment items were physically present and exercised but not the two SRIs, which were simulated by specifically developed software modules
- Electrical integration tests and "low-level " (bus communication) compliance tests of the On-Board Computer and the SRI, were performed with the actual SRI

Board Recommendations - 1

- **R1 N**o software function should run during flight unless it is needed.
- **R2** Prepare a test facility including as much real equipment as technically feasible, inject realistic input data, and perform complete, closed-loop, system testing. Complete simulations must take place before any mission. A high test coverage has to be obtained.
- **R3** Do not allow any sensor, such as the inertial reference system, to stop sending best effort data.
- **R4** Organize, for each item of equipment incorporating software, a specific software qualification review. The Industrial Architect report on complete system testing performed with the equipment.
  - All restrictions on use of the equipment shall be made explicit for the Review Board.
  - Make all critical software a Configuration Controlled Item (CCI).

Board Recommendations - 2

- **R5** Review all flight software (including embedded software), and in particular :
  - Identify all implicit assumptions made by the code and its justification documents on the values of quantities provided by the equipment. Check these assumptions against the restrictions on use of the equipment.
  - Verify the range of values taken by any internal or communication variables in the software.
  - Solutions to potential problems in the on-board computer software, paying particular attention to on-board computer switch over, shall be proposed by the project team and reviewed by a group of external experts, who shall report to the on-board computer Qualification Board.
- **R6** Wherever technically feasible, consider confining exceptions to tasks and devise backup capabilities.
- **R7** Provide more data to the telemetry upon failure of any component, so that recovering equipment will be less essential.
- **R8** Reconsider the definition of critical components, taking failures of software origin into account (particularly single point failures).

Board Recommendations - 3

- **R9** Include external (to the project) participants when reviewing. Make sure that these reviews consider the substance of arguments, rather than check that verifications have been made.
- **R10** Include trajectory data in specs and test requirements.
- **R11** Review the test coverage of existing equipment and extend it.
- **R12** Give the justification documents the same attention as code. Improve keeping code and its justifications consistent.
- **R13** Set up a team that will prepare the procedure for qualifying software, propose stringent rules for confirming such qualification, and ascertain that specification, verification and testing of software are of a consistently high quality in the Ariane 5 program.
- **R14** A more transparent organization of the cooperation among the partners in the Ariane 5 program. Engineering cooperation, with clear cut authority and responsibility, is needed for system coherence, with simple and clear interfaces between partners.

Maintenance Lessons Learned

- ✓ Decision makers need a good understanding of software's role in the enterprise
- ✓ Failure of all kinds will occur – plan for them
  - – Identify and manage risks
  - – Collect and utilize information on failures
  - – Use the level of fault tolerance risks warrant – a lot of it is cheap if planned from beginning
- ✓ Solutions to one problem can result in new ones

Maintenance Lessons Learned

- ✓ Do realistic testing even though it is hard
- ✓ Do substantive review of the substantive engineering decisions
  - – Do not let this be obscured by a mountain of paperwork

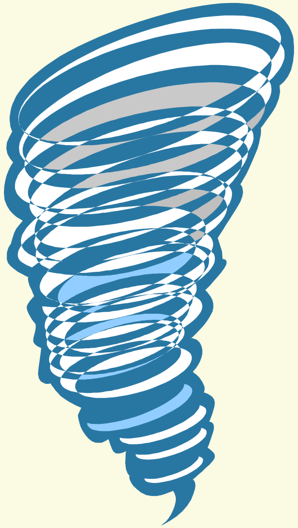
Lessons Learned

- ✓ The necessary information to avoid major problems is almost always available somewhere in the organization
- ✓ Be uncomfortable with comforting presumptions of “no impact”
- ✓ Unneeded software can create unneeded risk
- ✓ Agreement among all the organizations directly involved does not make it the right thing to do

Lessons Learned

- ✓ A history of a piece of software operating successful in the pre-change environment means it is (probably) qualified to operate in the pre-change environment
- ✓ Unthinkingly building software to the specifications is foolish
- ✓ Simple interfaces among organizations involved in a complex enterprise can only be the result of oversimplification
    - ✓ ***Reuse without a contract is sheer folly.***

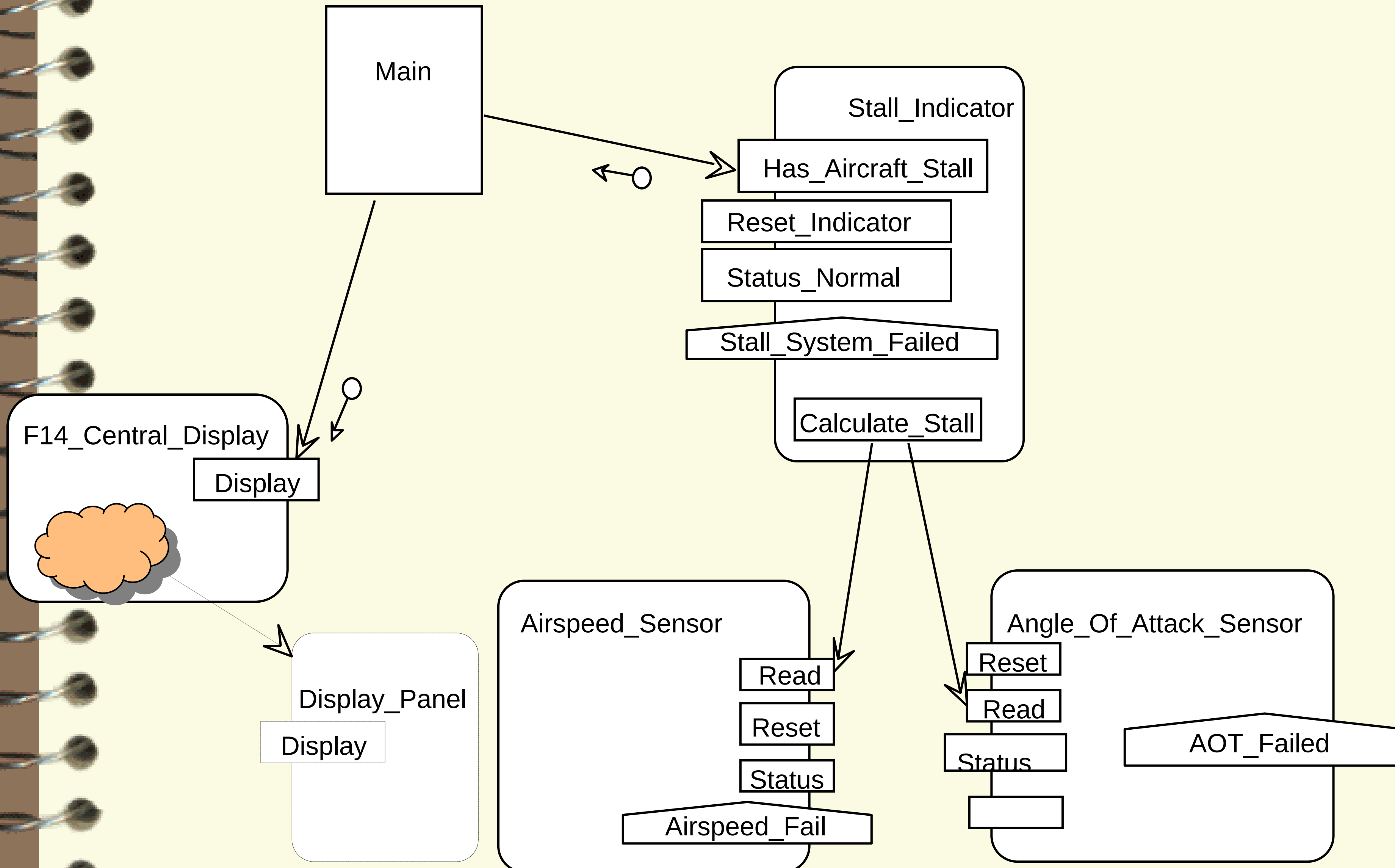
Whirlwind Tour
Of Ada 95



This section will….

- Give you a "quick and dirty" "firehose in the mouth" approach to Ada 95
- Discuss the building blocks of Ada programs

Anatomy of a Ada Program



Language Concepts

- ✓ Strong typing across separate compilation boundaries
- ✓ The "library" concept
- ✓ All identifiers declared before they're used
- ✓ "Context clauses" create links to pre-compiled library units

Strong Typing

Ada enforces strong typing using "name equivalence."

```
type APPLES is range 0 .. 100; -- an integer type
type ORANGES is range 0 .. 100; --ditto

A : APPLES;
O : ORANGES;
```

--> A and O are not compatible.

The Library

* All compilations are made in the context of a library.
* Identifiers not declared in the current compilation unit must have been pre-compiled into the library.
* All compilations have access to the identifiers declared in package STANDARD which defines several common types and operations on those types.
* The LRM also defines several other packages, most notably Ada.Text_IO, which every Ada system provides.
* It is up to the programmer to explicitly include any package, except ADA, in a compilation.
* The following user defined program units are compilable into the library:
  ** Subprograms
  ** Packages
  ** Generics
  ** Generic Instantiations

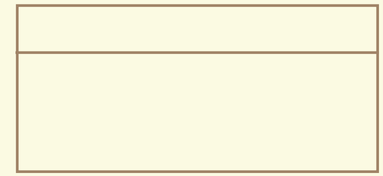
Context Clauses

A context clause is the mechanism a programmer uses to introduce pre-compiled identifiers into the current compilation.

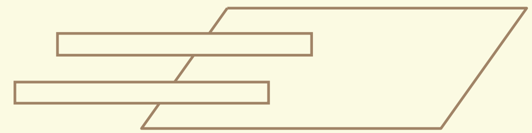
```
with Ada..Text_IO;       -- The context clause
procedure HELLO is
      begin
         Ada.Text_IO.PUT_LINE("Hello, World!");
      end HELLO;
```

Program Units

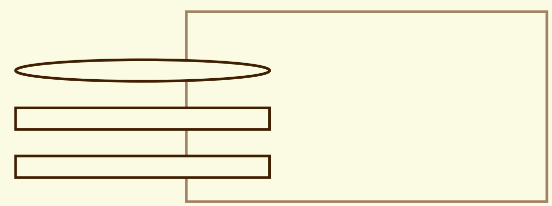
* Subprograms
  -- Functions and Procedures
  -- Main Program
  -- Abstract Operations



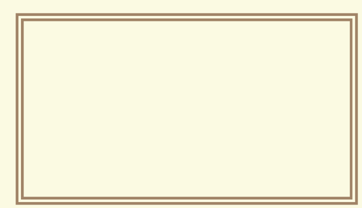
* Tasks
  -- Parallel Processing
  -- Real-time Programming
  -- Interrupt Handling



* Packages
  -- Encapsulation
  -- Information Hiding
  -- Abstract Data Types



* Generics
  -- Templates



All Ada Program Units

The SPECIFICATION is the contract or interface between the user of the unit and the implementor of the unit. It represents "what" is to be done, not "how". Similar in concept to:

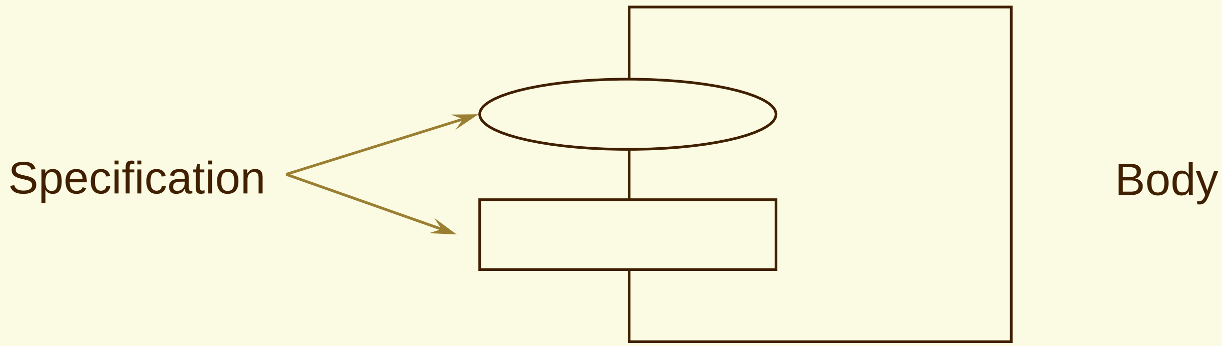
Pascal's forward and

C's prototype

The BODY is the "how" of the unit. Its details are the responsibility of the implementor. The user of the unit does not have access to these details.

Separate compilation -- the body of any unit can be separately compiled from its specification, as long as the specification is compiled first.

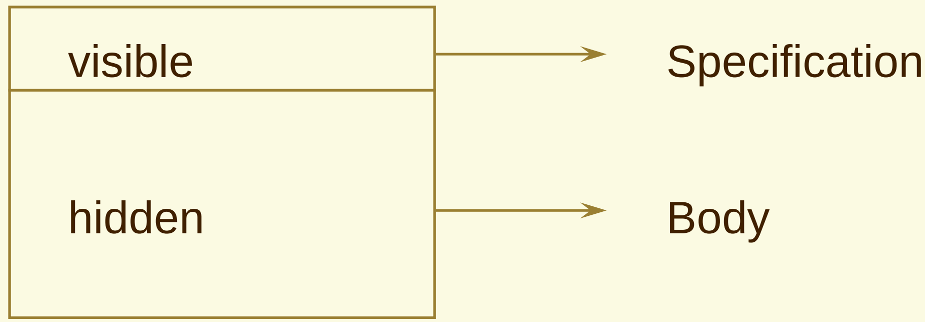
Subprogram Units



"What" the program unit does ← **ABSTRACTION** → "How" the program unit does what it does

All the user of the program unit needs to know ← **INFORMATION HIDING** → The details of implementation are inaccessible to the user

Ada Subprograms



- Procedures
  - -- Perform some "sub-actions"
  - -- Call always appears as a statement
- Functions
  - -- Calculate and return a value
  - -- Call always appears in an expression

Procedures

```
-- Procedure Specification
procedure SWAP (PRE, POST: in out INTEGER);

-- Procedure Body
procedure SWAP (PRE, POST: in out INTEGER) is
     TEMP: INTEGER := PRE;
begin
     PRE := POST;
     POST := TEMP;
end SWAP;

-- Procedure Call
SWAP (MY_COUNT, YOUR_COUNT);
SWAP (MY_COUNT, POST => YOUR_COUNT);
SWAP  (PRE => MY_COUNT, POST =>  YOUR_COUNT);
```

Ada Function

-- Function Specification

```
function SQRT (ARG:  FLOAT) return FLOAT;
```

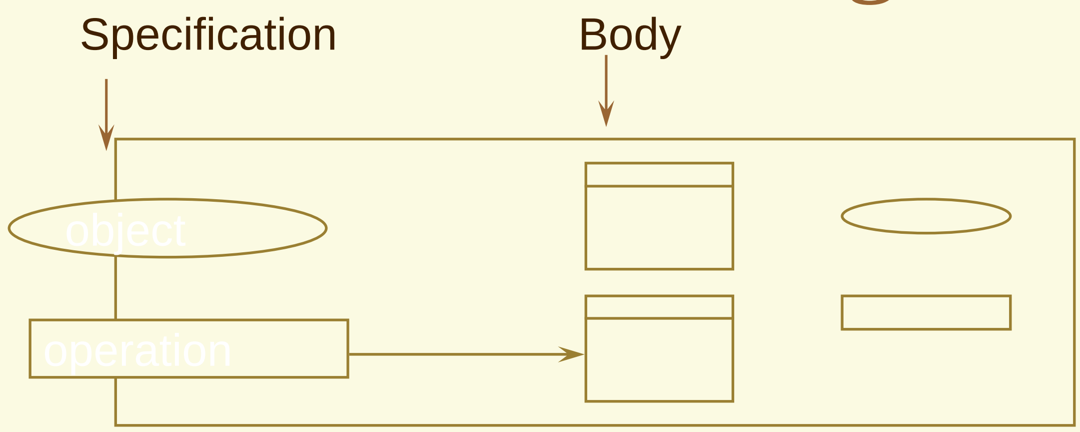
-- Function Body

```
function SQRT (ARG:  FLOAT) return FLOAT is
     RESULT:  FLOAT;
begin
     -- algorithm for computing RESULT goes here
     RETURN RESULT;
end SQRT;
```

-- Function Call (Assumes STANDARD_DEV and VARIANCE are of type FLOAT)

```
STANDARD_DEV := SQRT (VARIANCE);
```

Ada Packages



* The PACKAGE is the primary means of "extending" the Ada language.

* The PACKAGE hides information in the body thereby enforcing the abstraction represented by the specification.

* Operations (subprograms, functions, etc) whose specification appear in the package specification must have their body appear in the package body.

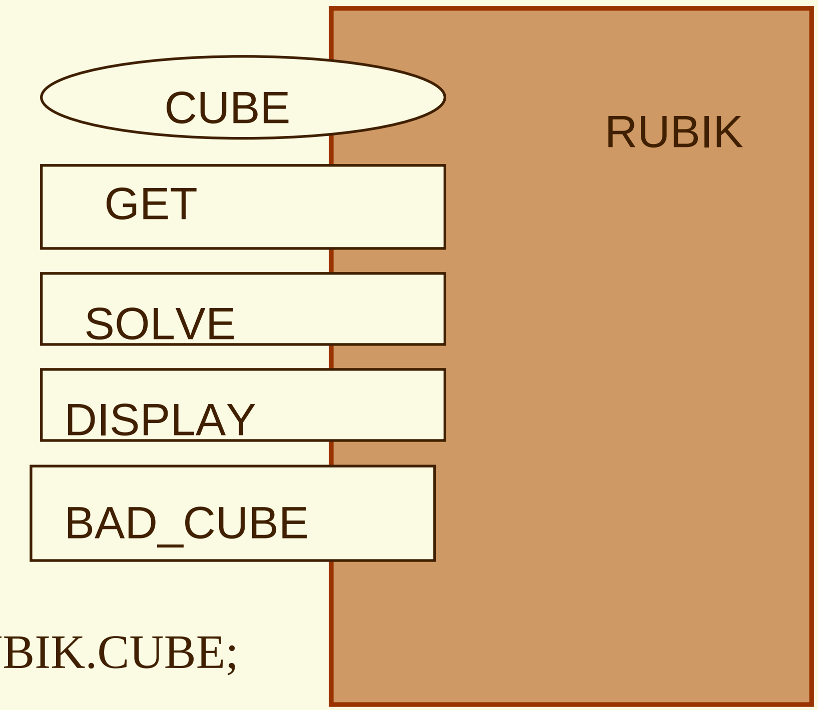
* Other units (subprograms, functions, packages, etc) as well as other types, objects etc may also appear in the package body. If so, they are not visible outside the package body.

Ada Packages

```
-- Package Specification
package RUBIK is
    type CUBE is private;
    procedure GET (C : out CUBE);
    procedure SOLVE (C : in out CUBE);
    procedure DISPLAY (C : in CUBE);
    BAD_CUBE : exception;
private
    type CUBE is . . .
end RUBIK;

-- Package Body
package body RUBIK is
    -- all bodies of subprograms found in the
    -- package spec go here along with any
    -- other local declarations that should
    -- be kept "hidden" from the user
    procedure GET (C : out CUBE) is . . .
    procedure SOLVE (C : in out CUBE) is . . .
    procedure DISPLAY (C : in CUBE) is . . .
end RUBIK;
```

Package Usage

MAIN

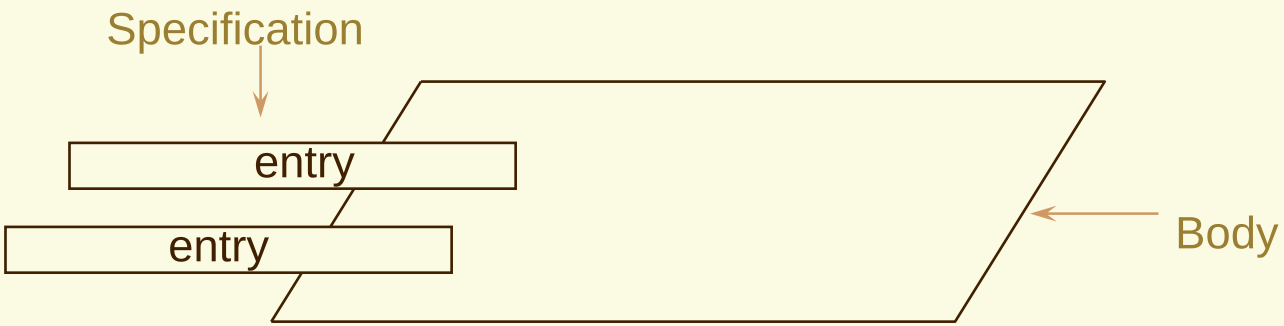



```
with RUBIK;
with Ada.Text_IO;
procedure MAIN is
    MY_CUBE :  RUBIK.CUBE;
begin
    RUBIK.GET(MY_CUBE);
    RUBIK.SOLVE(MY_CUBE);
    RUBIK.DISPLAY(MY_CUBE);
exception
    when RUBIK.BAD_CUBE =>
            Ada.Text_IO.PUT_LINE("You've got a bad one");
end MAIN;
```

Example

```
Package MEASURES is
      type AREA is private;
      type LENGTH is private;
      function "+" (LEFT, RIGHT : LENGTH) return
      LENGTH;
      function "*" (LEFT, RIGHT : LENGTH) return AREA;
private
      type LENGTH is range 0..100;
      type AREA is range 0..10000;
end MEASURES;
------------------------------------------------------------------
      ------
with MEASURES;  uses MEASURES;  --direct visibility
procedure MEASUREMENT is
use MEASURES.Length;
use MEASURES.Area;;  --direct visibility
      SIDE1,SIDE2  :  LENGTH;
      FIELD :  AREA;
begin
      …...
      FIELD := SIDE1 * SIDE2
end MEASUREMENT;
```

Ada Tasks



* The TASK concept in Ada provides a model of parallelism which encompasses

  -- multicomputers

  -- multiprocessors

  -- interleaved execution

* In Ada, the method of communication between tasks is known as "rendezvous"

* Ada "draws up" into the language certain capabilities previously performed only by the operating system

Task Communication

```
--  Task Specification
    task TASK_1;        --  no entries
    task TASK_2 is
            entry XMIT (N : INTEGER);
    end TASK_2;

--  Task Bodies
    task body TASK_1 is
            TASK_2.XMIT(17);      --  an entry call
    end TASK_1;

    task body TASK_2 is
            accept XMIT (N : INTEGER) do
            --  statements to be executed during
            --  rendezvous
            end XMIT;
    end TASK_2;
```

Tasks

simpliest form:
no communication

task ALWAYS belongs to someone (MAIN)

MAIN cannot end until all tasks end

execution of T1 and T2 begin here

```
procedure MAIN is
        task T1;
        task T2;

        task body T1 is
        begin
              null;
        end T1;

        task body T2 is
        begin
              null;
        end T2;

begin  --Main
      null;
end MAIN;
```

Tasking Example

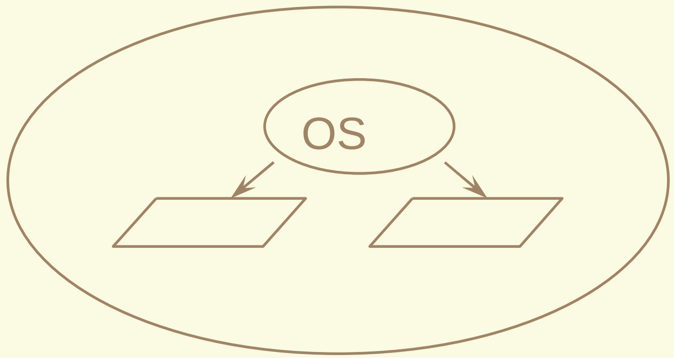
task specs

task bodies

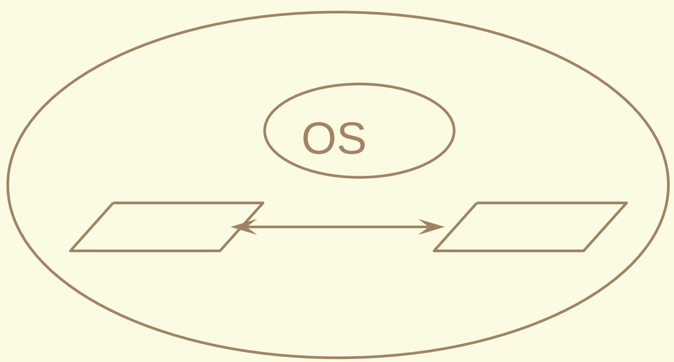
main program

```
with Ada.Text_IO;
procedure TASK_EXAMPLE is
    task PLAIN;

    task WITH_LOCAL_DECLARATION;


    task body PLAIN is
    begin
            null;
    end PLAIN;

    task body WITH_LOCAL_DECLARATION is
            FOREVER : constant STRING :=
                    "forever";
    begin
            loop
                    Ada.Text_IO.PUT("This prints");
                    Ada.Text_IO.PUT_LINE(Forever);
            end loop;
    end WITH_LOCAL_DECLARATION;

begin
    null;
end TASK_EXAMPLE;
```

The Ultimate in Information Hiding



THE
TRADITIONAL
MODEL OF
CONCURRENCY



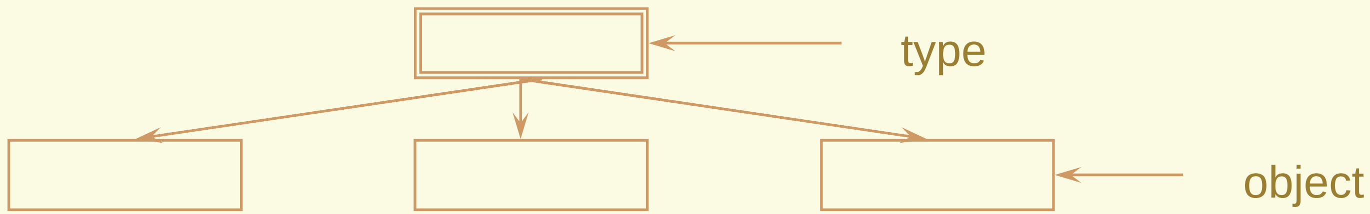
The Ada
TASKING
MODEL

Ada Types

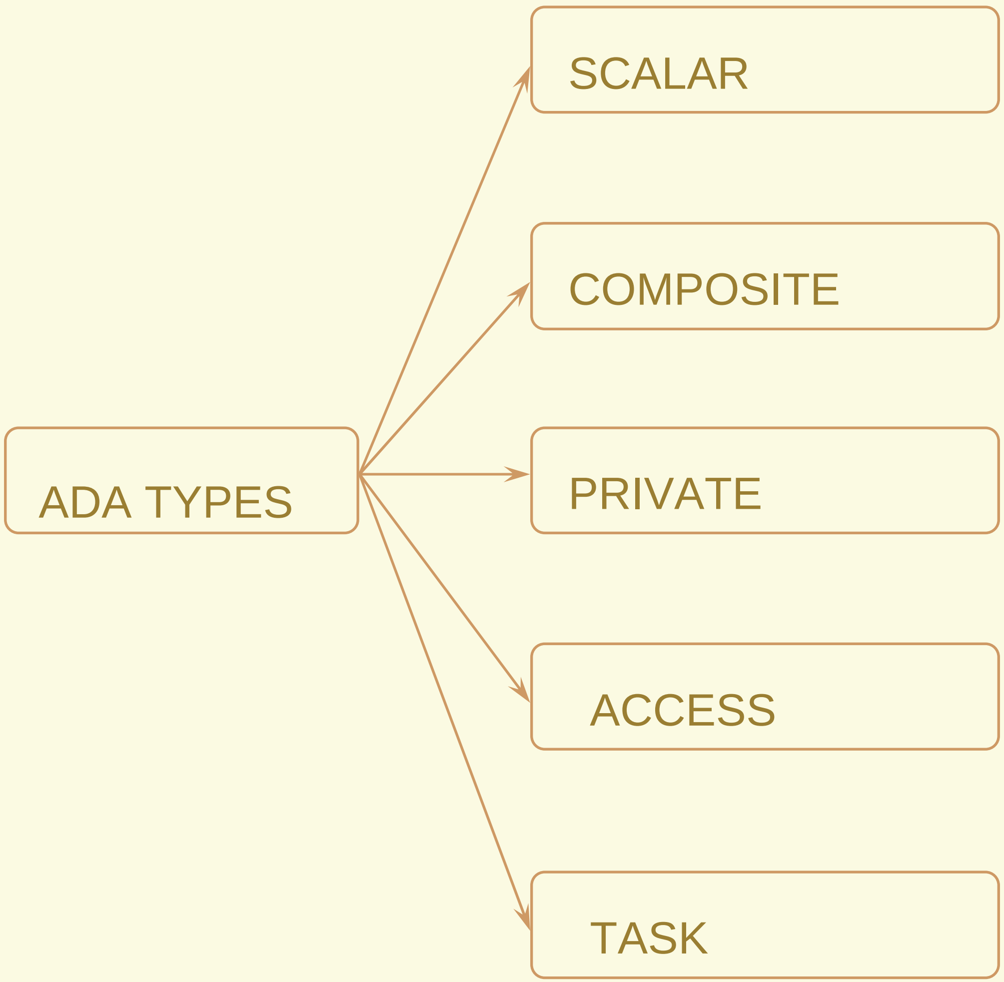
* An Ada type is a template for objects; it represents a set of values which are meaningful for the objects and also a set of operations on the objects (values)

* Ada is a strongly typed language. This means that all objects must be declared and objects of different types cannot be implicitly mixed in operations

* TYPES are not operated upon directly. They are a means of declaring instances called OBJECTS. These objects can be operated upon



Ada Types



objects are
single values

objects contain
other components

objects are
'abstract'

objects 'point' to
other objects

objects are
parallel processes

Ada Types

SCALAR TYPES			
DISCRETE		REAL	
INTEGER	ENUMERATED	FIXED	FLOAT

integer	boolean	duration	float
natural	character		
positive			
long_integer			
short_integer			

Integer Types

* An Integer type characterizes a set of whole number values and a set of operations on whole numbers

```
type DEPTH is range - 1000 .. 0;
type ROWS is range 1 .. 8;
type LINES is range 0 .. 66;

subtype TERMINAL is LINES range 0 .. 24;
```

<u>ELEMENT OBJECT DECLARATIONS</u>

```
ROW_COUNT  :  ROWS;
LINE_COUNT  :  LINES := 1;
CRT           :  TERMINAL := 16;
FATHOMS      :  constant DEPTH :=-100;
```

ROW_COUNT	LINE_COUNT	CRT	FATHOMS
undef	1	16	-100

Enumeration Types

ENUMERATION TYPE DECLARATIONS

```
type COLOR is (WHITE, RED, YELLOW, GREEN, BLUE);
type LIGHT is (RED, AMBER, GREEN);
type GEAR_POSITION is (UP, DOWN, NEUTRAL);
type SUITS is (CLUBS, DIAMONDS, HEARTS, SPADES);
subtype MAJORS is SUITS range HEARTS..SPADES;
type BOOLEAN is (FALSE, TRUE);          -- predefined
```

ENUMERATION OBJECT DECLARATIONS

```
HUE : COLOR;
SHIFT : GEAR_POSITION :=GEAR_POSITION'last;
T : constant BOOLEAN := TRUE;
HIGH : MAJORS :=CLUBS; -- invalid
```

HUE	SHIFT	T
undef	NEUTRAL	TRUE

Character Types

<u>CHARACTER TYPE DECLARATIONS</u>

```
type ROMAN_DIGIT is ('I', 'V', 'X', 'L', 'C', 'D', 'M');

type VOWELS is ('A', 'E', 'I', 'O', 'U');

subtype FORTRAN_CONVENTION is CHARACTER
                                         range 'I' .. 'N';
```

<u>CHARACTER OBJECT DECLARATIONS</u>

```
INDEX : FORTRAN_CONVENTION :='K';
ROMAN_100 : constant ROMAN_DIGIT := 'C';
MY_CHAR : CHARACTER;
```

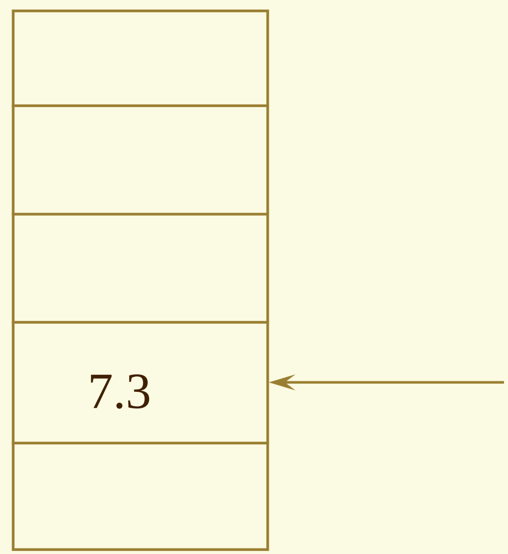
INDEX	ROMAN_100	MY_CHAR
'K'	'C'	UNDEF

Constrained Arrays

type TABLE is array(INTEGER range 1 .. 5) OF FLOAT;

MY_LIST : TABLE := (3.7, 14.2, -6.5, 0.0, 1.0);

MY_LIST (4) := 7.3;



MY_LIST

Constrained Arrays

```
type DAYS is (SUN, MON, TUE, WED, THU, FRI, SAT);
type WEEK_ARRAY is array (DAYS) of BOOLEAN;

MY_WEEK : WEEK_ARRAY := (MON .. FRI => TRUE, others => FALSE);
```

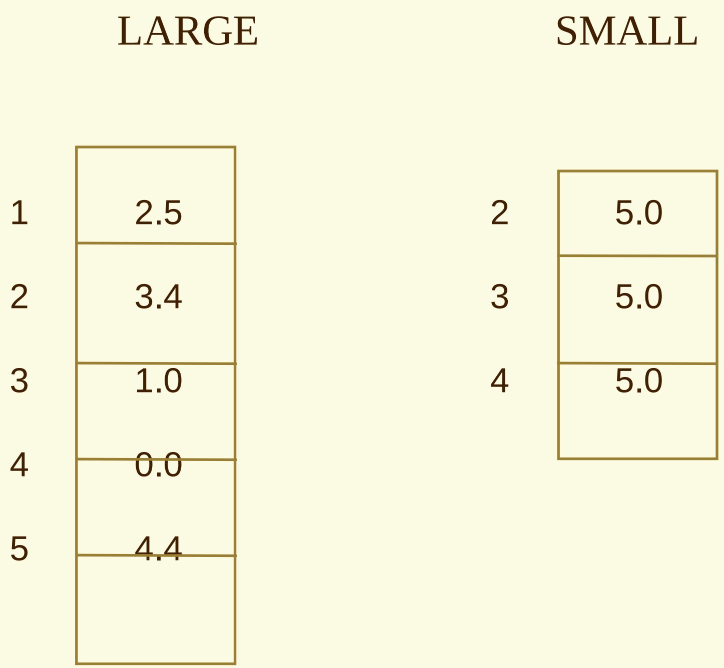
SUN	FALSE
MON	TRUE
TUE	TRUE
WED	TRUE
THU	TRUE
FRI	TRUE
SAT	FALSE

MY_WEEK

Unconstrained Arrays

* INDEX TYPE AND COMPONENT TYPE BOUND TO ARRAY TYPE
* INDEX RANGE BOUND TO OBJECTS, NOT TYPE
* ALLOWS FOR GENERAL PURPOSE SUBPROGRAMS
* INCLUDES Ada STRING TYPE

```
Type SAMP is array (INTEGER range <>) of FLOAT;
LARGE : SAMP (1 .. 5) := (2.5, 3.4, 1.0, 0.0, 4.4);
SMALL : SAMP (2 .. 4) := (others => 5.0);
```



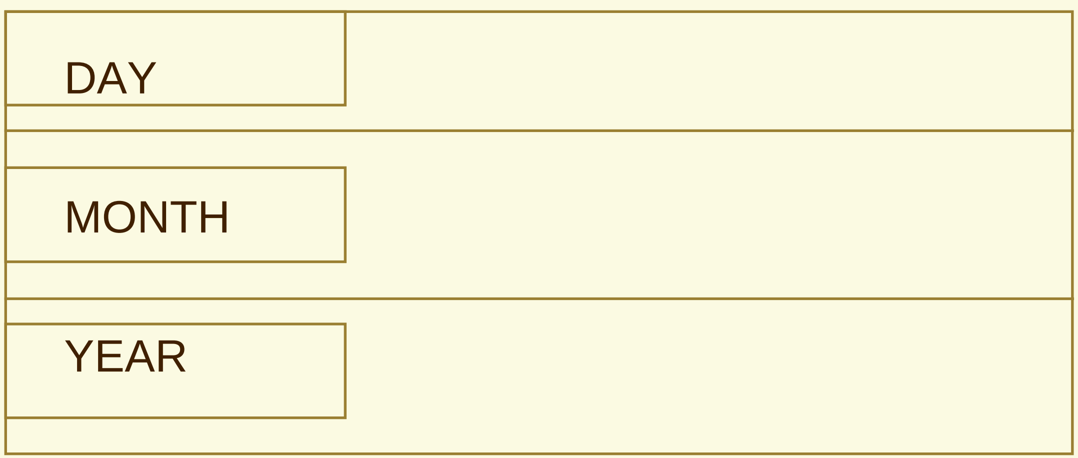
Records

<u>RECORD TYPE DECLARATION</u>

```
type DATE is record
            DAY : INTEGER range 1 .. 31;
            MONTH : MONTH_TYPE;
            YEAR : INTEGER range 1700 .. 2150;

      end record;
```

<u>RECORD OBJECT DECLARATION</u>

```
TODAY : DATE;
```



Record Type

RECORD TYPE DECLARATION

Record types may have default values for the components.

```
type DATE is record
        DAY : INTEGER range 1 .. 31 := 17;
        MONTH : MONTH_TYPE := January;
        YEAR : INTEGER range 1700 .. 2150
            := 1982; --year I was born
    end record;
```

RECORD OBJECT DECLARATION

```
TODAY : DATE;
```

Record Type

RECORD COMPONENT REFERENCE

```
TODAY.DAY :=13;
TODAY.MONTH :=JUN;
TODAY.YEAR :=1990;
```

RECORD OBJECT REFERENCE

```
TODAY := (13,JUN,1990);  -- an aggregate

        -- or --

if TODAY /= (6, DEC, 1942) then ...
```

Strings

```
type STRING is array (natural range <>) of character;
STR_5 : string (1 .. 5);
STR_6 : string (1 .. 6) := "Framus";
WARNING :constant STRING := "DANGER";
```

STR_6

F	r	a	m	u	s

WARNING

D	A	N	G	E	R

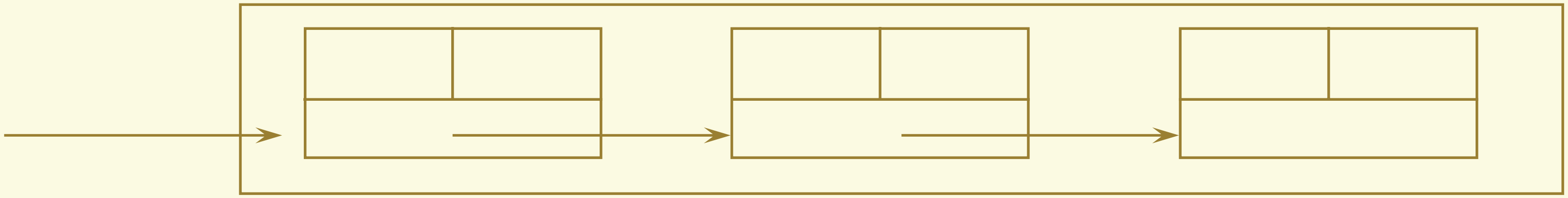
```
subtype TEN_LONG is STRING (1 .. 10);
FIRST_TEN : TEN_LONG := "HEADER    ";
```

FIRST_TEN

H	E	A	D	E	R				

Access Types

```
type NODE;
type PTR is access NODE;
type NODE is
record
    FIELD1 : SOME_TYPE;
    FIELD2 : SOME_TYPE;
    FIELD3 : PTR;
end record;
```



```
TOP :PTR; -- an access object

TOP := new NODE; -- an allocator

TOP.FIELD3 :=new NODE; -- another allocator
```

Private Types

* ACTUAL TYPE DESCRIPTION IS "HIDDEN"
* THE TYPE IS PRIMARILY KNOWN THRU ITS OPERATIONS
* PRIVATE TYPES ARE ALWAYS IMPLEMENTED BY PACKAGES
* PRIVATE TYPES PROTECT DATA FROM ERRONEOUS ACCESS
* IF AN OBJECT IS OF A PRIVATE TYPE, ASSIGNMENT, (IN) EQUALITY AND ALL EXPLICITLY DECLARED OPERATIONS ARE ALLOWED
* IF AN OBJECT IS OF A LIMITED PRIVATE TYPE, ONLY THE EXPLICITLY DECLARED OPERATIONS ARE ALLOWED

Statements

SEQUENTIAL	CONDITIONAL	ITERATIVE
ASSIGNMENT	IF	LOOP
NULL	-- THEN	-- EXIT
PROCEDURE CALL	-- ELSE	-- FOR
RETURN	-- ELSIF	-- WHILE
BLOCK	CASE	

TASKING	OTHER
DELAY	RAISE
ENTRY CALL	
ABORT	CODE
ACCEPT	
SELECT	goto

Assignment Statements

VARIABLE := EXPRESSION ;

* The variable takes on the value of the expression

* The variable and the expression must be of the same type

```
MY_INT := 17;                  -- Integer

LIST(2..4) :=LIST (7..9);      -- slice

TODAY := (13,JUN,1990);        -- aggregate

X := SQRT (Y);                 -- function call
```

Conditionals (IF)

```
if TODAY = (30, JUL, 1943) then
    PEGS_YEARS := PEGS_YEARS + 1;          --NOTE - NO “begin
    GET (BIRTHDAY_CARD);                          --end” needed for
end if;                                    --multiple lines

if IS_ODD (NUMBER) then
    ODD_TOTAL := ODD_TOTAL + 1;
else
    EVEN_TOTAL := EVEN_TOTAL + 1;
end if;

if    SCORE >= 90 then GRADE := 'A';
elsif SCORE >= 80 then GRADE := 'B';
elsif SCORE >= 70 then GRADE := 'C';
elsif SCORE >= 60 then GRADE := 'D';
else                   GRADE := 'F';
end if;
```

Conditionals (CASE)

```
procedure SWITCH (HEADING :in out DIRECTION) is
   begin
    case HEADING is
        when NORTH => HEADING := SOUTH;
        when EAST      => HEADING := WEST;
        when SOUTH  => HEADING := NORTH;
        when WEST    => HEADING := EAST;
    end case;
   end SWITCH;
```

```
case NUMBER is
   when 2          => DO_SOMETHING;
   when 3 | 7 | 8 => DO_SOMETHING_ELSE;  -- '|' means or
   when 9 .. 20   => DO_SOMETHING RADICAL;
   when others   => PUNT;                          --everything else
end case;
```

NOTE that a case statement must be exclusive and exhaustive.
This is why the "others" is necessary.

Iteration (Loops)

```
loop
    GET_SAMPLES;
    exit when EXHAUSTED;
    PROCESS_SAMPLES;
end loop;
```

```
for i in 1..5 loop          --'i' is defined implicitly  It does not
    GET_IT(NUMBER);-- exist outside of the loop.
    MODIFY_IT (NUMBER);
end loop;
```

```
while DATA_REMAINS loop
    <sequence_of_statements>
end loop;
```

Sample Program

```
with Ada.Text_IO;
procedure Second_Sample_Program is
        type       SUM_ELEMENT_TYPE is  range -2000..2000;
        subtype INPUT_ELEMENT_TYPE is SUM_ELEMENT_TYPE range -1000..1000;
        A_NUMBER : INPUT_ELEMENT_TYPE : 0;
        B_NUMBER : INPUT_ELEMENT_TYPE := 0;
        C_NUMBER : SUM_ELEMENT_TYPE  := 0;

        ----------- Instantiation -------------------------
        package NEW_INTEGER_IO is  new Ada.Text_IO.INTEGER_IO(SUM_ELEMENT_TYPE);
        ----------------------------------------------------
begin
        Ada.Text_IO.PUT_LINE("Enter Integer -1000 to 1000");
        NEW_INTEGER_IO.GET(A_Number);
        Ada.Text_IO.PUT_LINE("Enter Integer -1000 to 1000");
        NEW_INTEGER_IO.GET(B_Number);

        C_NUMBER := A_NUMBER + B_NUMBER;
        Ada.Text_IO.NEW_LINE;
         NEW_INTEGER_IO.PUT(A_NUMBER);
        Ada.Text_IO.PUT(" + ");
        NEW_INTEGER_IO.PUT(B_NUMBER);
        Ada.Text_IO.PUT(" = ");
        NEW_INTEGER_IO.PUT(C_NUMBER);

        Ada.Text_IO.NEW_LINE;

end SECOND_SAMPLE_PROGRAM;
```

Sample Program

```
    --  To exemplify some of the Ada statements, let's look
    --  at the implementation of a 'wrap-around'successor
    --  function for type DAYS.

procedure TEST is

    type DAYS is (SUN, MON, TUE, WED, THU, FRI, SAT);

    TODAY  :  DAYS;          -- Object declarations
    TOMORROW  :  DAYS;

    function WRAP (D : DAYS) return DAYS is . . .

begin
    . . .
    TOMORROW := WRAP(TODAY);
    . . .
end TEST;
```

Sample Program

```
function WRAP (D : DAYS) return DAYS is
begin
    case D is
      when SUN => return MON;
      when MON => return TUE;
      when TUE => return WED;
      when WED => return THU;
      when THU => return FRI;
      when FRI => return SAT;
      when SAT => return SUN;
    end case;
end WRAP;
```

Sample Program

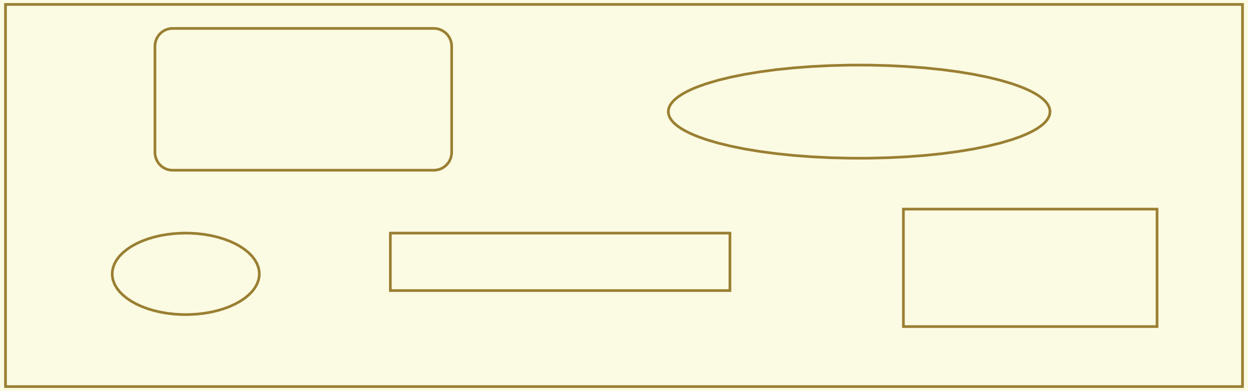
```
function WRAP ( D : DAYS ) return DAYS is
begin
    return DAYS'SUCC(D);
exception
    when CONSTRAINT_ERROR =>
        return DAYS'FIRST;
end WRAP;
```

Generics

GENERICS DEFINE

A TEMPLATE OR

MOLD FOR PROGRAM UNITS



Generics

Dictionary Definition:

a. Relating to, or characteristic of, a whole group or class

b. General

* Defines a template for a general purpose program units
* The template must be instantiated prior to use in a program
* Strong typing makes generics a necessary language feature

Why Generics?

Generics are like high-level macros. We parameterize them at compile time because we can't pass types or subprograms at execution time. Parameters to generics can be

types

subprograms

objects

values

packages

With a generic, we can pass a type and reuse one general routine or algorithm for a variety of data types.

Non-Generic Swap

```
procedure Integer_Swap(Integer_1 :  in out Integer;
                       Integer_2 :  in out Integer) is
  Temp : Integer := 0;
begin
  Temp := Integer_1;
  Integer_1 := Integer_2;
  Integer_2 := Temp;
end Integer_Swap;
```

PROBLEMS?? It only works for integers.

Generic Swap

generic formal part

generic paramenter

spec

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

body

```
procedure SWAP(ITEM_1 : in out ELEMENT;
               ITEM_2 : in out ELEMENT) is
  TEMP : ELEMENT;
begin
  TEMP   := ITEM_1;
  ITEM_1 := ITEM_2;
  ITEM_2 := TEMP;
end Swap;
```

Using the Generic

```
with Swap;
procedure Example is
    procedure Int_Swap is new Swap(Integer);
    procedure Chr_Swap is new Swap(Character);
    Num1, Num2 :  Integer;
    Char1, Char2 :  Character;
begin
    Num1 := 10;
    Num2 := 3;
    Int_Swap (Num1, Num2);

    Char1 := 'A';
    Char2 := 'Z';
    Chr_Swap (Char1, Char2);
end Example;
```

Summary of the “Quick-and-Dirty” Overview

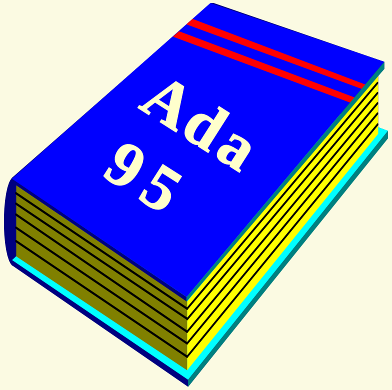
* **Subprograms**
* **Packages**
* **Exception Handlers**
* **Tasks**
* **Types and Objects**
* **Statements**
* **Generics**

Homework

- **Learn to use the Ada compiler**
- **Write a simple program that prints out a simple message to you**

Language Concepts

- ✓ Strong typing across separate compilation boundaries
- ✓ The "library" concept
- ✓ All identifiers declared before they're used
- ✓ "Context clauses" create links to pre-compiled library units



What makes Ada unique?

- ✓ Emphasis on strong typing
- ✓ Use of library for checking compilation
  - All interfaces checked during compilation process
  - For many languages, the most important tool is the debugger
  - For Ada, the most important tool is the compiler

"The tools we use have a profound (and devious) influence on our thinking habits, and therefore, on our thinking abilities."

--E.W. Dijkstra

Strong typing

- ✓ Ada enforces strong typing using "name equivalence." (not structural equivalence)
  - Applies to both predefined and user-defined types

```
procedure TESTER is
    type APPLES is range 0 .. 100; -- an integer type
    type ORANGES is range 0 .. 100; --ditto
    A : APPLES := 100;              - - initialize during declaration
    O : ORANGES;
begin
    O := A;                         - - ERROR, incompatible types
end;
```

--> A and O are not compatible.

Benefits of OOD for Complex Systems

- An OOD approach facilitates
- Identifying and creating software modules with strong internal cohesion and weak external coupling with other modules.
- Identifying and creating Ada packages.
- Providing a direct correspondence between the physical world and the computer software.

Disadvantages of OOD for Complex Systems

- OOD may have negative space and timing impact.
- OOD requires replacing the traditional functional
  decomposition mindset.

What is an Object?

- A self-contained thing which is characterized by:
  - it's states
  - the actions it performs
  - the actions that can be performed upon it
  - an unique designation
- An Object is an instance of a Class (set,type,group)
- Views of an Object:
  - External perspective
  - Internal perspective
- An Object may be composed of one or more Objects

Ada Object

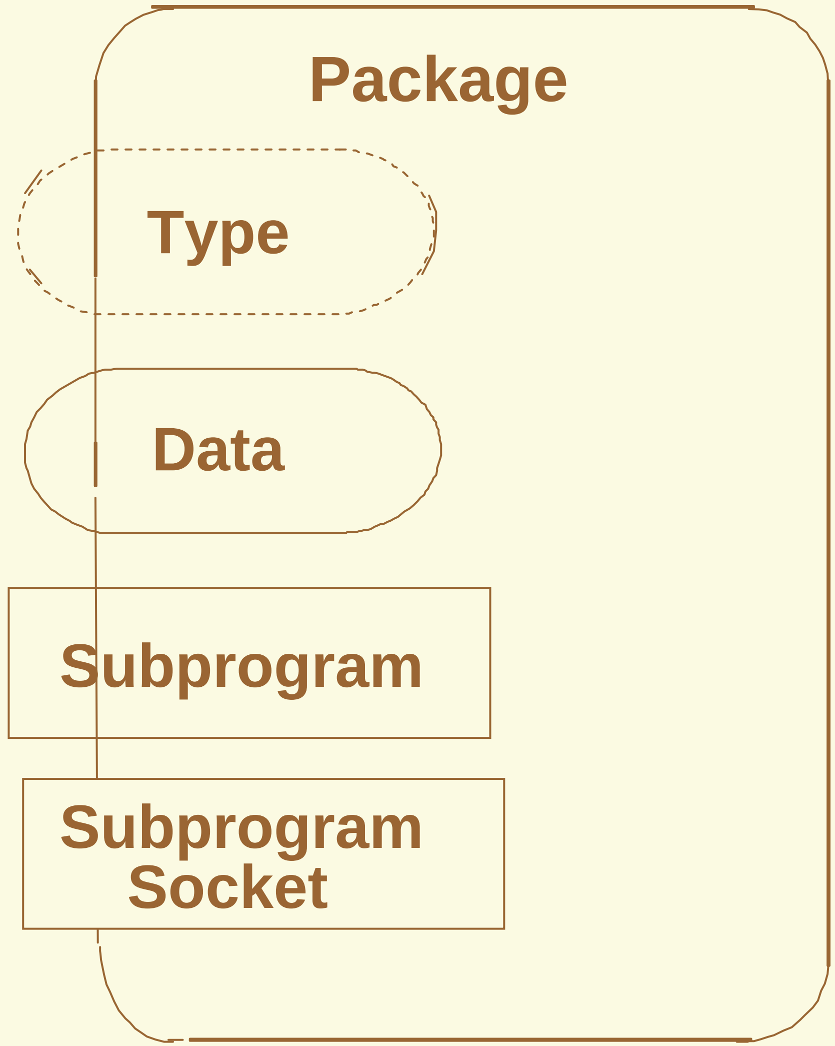
- In Ada, an Object is typically implemented as an:
- Abstract State Machine (ASM) package or generic package.
- Abstract Data Type (ADT) package or generic package.

What is an ASM?

- An ASM package or generic package encapsulates:
- A single, possibly complex Ada object of a single type, both typically hidden in the body
  Note: It may also encapsulate hidden auxiliary Ada objects and types.
- Exported operations (e.g. subprograms or task entry points) that operate on this object.
  Note: It may also encapsulate hidden auxiliary operations.
- Exceptions (error conditions) that are raised (and must be properly handled) when the operations cannot produce a valid result.
- An ASM is represented by a singular noun (e.g., Account_Payable).

Buhr Sample 1

Abstract State Machine



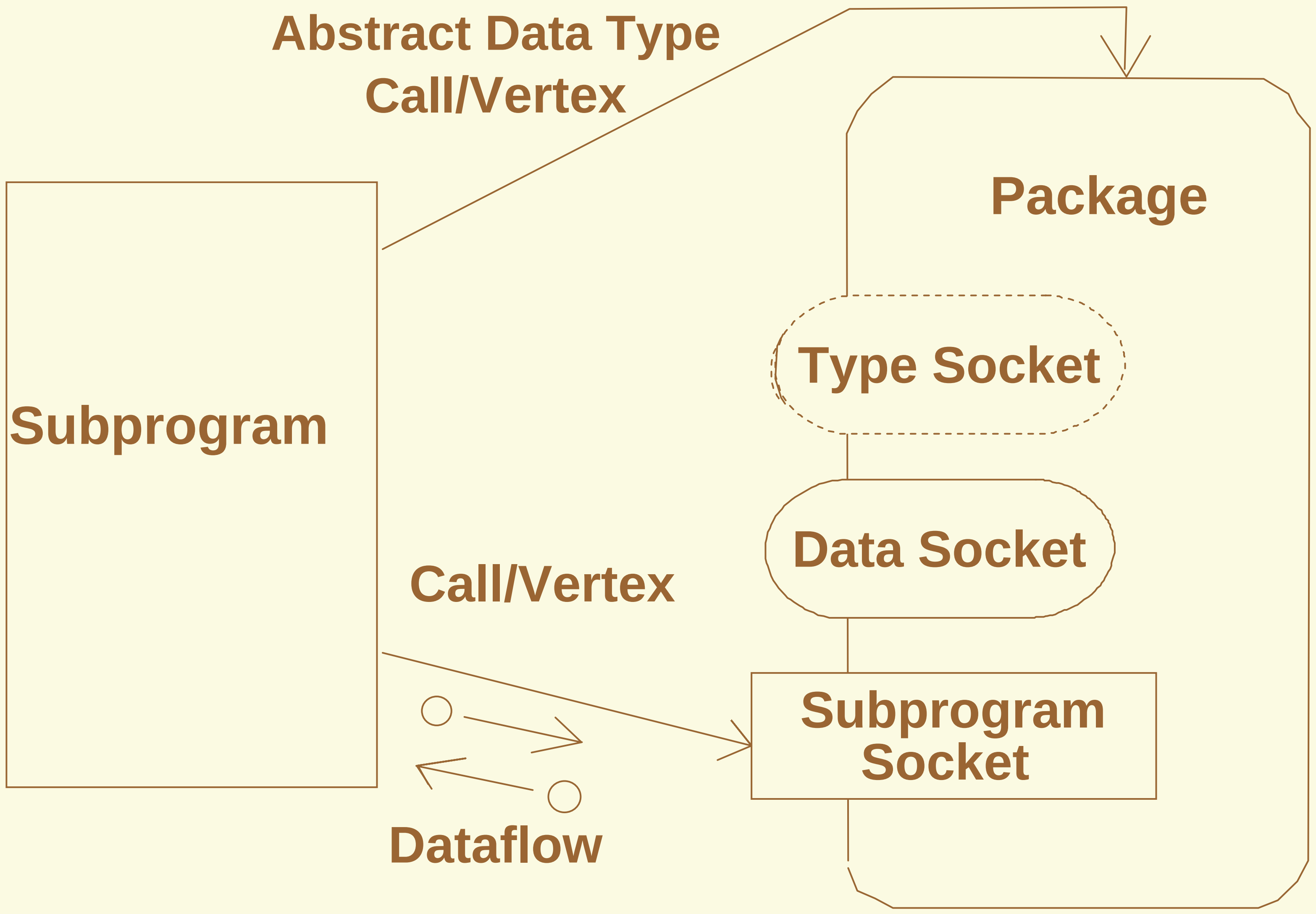
Example of an Abstract State Machine

```
with Aircrew_Characteristics;
package Aircrew is
- Abstract State machine
procedure Select_Crew_Member (Crew_Member : in
              Aircrew_Characteristics.Crew_Members);
procedure Swap_Crew_Member (Old_Crew_Member : in
              Aircrew_Characteristics.Crew_Members;
              New_Crew_Member : in
              Aircrew_Characteristics.Crew_Members);
procedure Select_Crew_For (Aircraft : in
              Aircrew_Characteristics.Aircraft_Class);
procedure Build_Crew;
function Numbers_Of_Crew_Members return Natural;
Can_Not_Build_Crew       : exception;
Crew_Member_Does_Not_Exist : exception;
end Aircrew;
with Aircraft_Classes;
package Aircrew_Characteristics is
  type Crew_Member is limited private;
  -- private part ...
end Aircrew_Characteristics;
```

What is an ADT?

- An ADT package or generic package encapsulates:
- Multiple export Ada objects (data variables) of a single type.
  Note: It may also encapsulate hidden auxiliary Ada objects and types.
- Exported operations (e.g. subprograms or task entry points) that operate on these objects.
  Note: It may also encapsulate hidden auxiliary operations.
- Exceptions (error conditions) that are raised (and must be properly handled) when the operations cannot produce a valid result.
- An ADT is represented by a plural noun (e.g. Windows)

Buhr Sample 2



Example of an Abstract Data Type

```
with Coordinates, Velocity;
package Targets_Id is
  type TARGETS is private;
  type TARGET_CLASSIFICATION is (FRIENDLY, ENEMY, UNKNOWN);
  type TARGET_FORM is (AIRCRAFT, MISSILE, UNKNOWN);

procedure Classify (Target : in out TARGETS);
procedure   Create (Target :    out TARGETS);
procedure   Delete (Target : in     TARGETS);
function Classification_Of (Target : in TARGETS)
                        return TARGET_CLASSIFICATION;
function Coordinates_Of    (Target : in TARGETS)
                   return Coordinates.COORDINATES_TYPE;
function Exists (Target : in TARGETS) return BOOLEAN;
function Form_Of (Target : in TARGETS) return TARGET_FORM;
function Velocity_Of (Target : in TARGETS) return
                              Velocity.VELOCITY_TYPE;
CAN_NOT_CREATE : exception; -- Create_Next
DOES_NOT_EXIST : exception; -- Classify, Delete, Form_Of
                        -- Classification_Of
private -- Hidden definition of target identification
  -- private part.  We don't care!
end Targets_Id;
```

Kinds of Objects

- Active
- Acts Autonomously
- Acts upon other objects
- May be acted upon
- Passive
- Never acts autonomously
- May act upon other objects
- Is acted upon

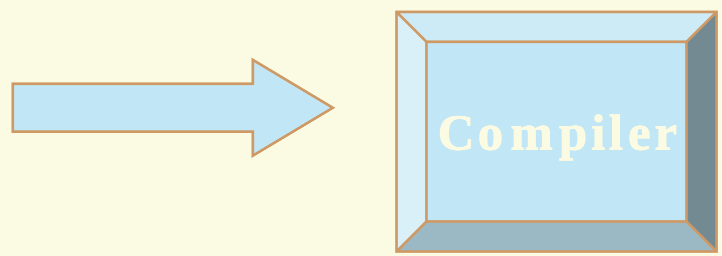
Classifications of Operations in OOD

- Constructors: Operations that change the state of an object (e.g. Push (Element_On,My_Stack);).
- Selectors: Operations that retrieve the state of an object (e.g. Is_Empty --Return true if stack is empty).
- Iterators: For an object that is a collection of other objects, operations that visit the collection's components. (e.g. Get_Next(The_Iterator : in out Iterator); - Advance the iterator to the next item in the queue).
- Desired characteristics of object operations
- Sufficient to permit all common uses of the object
- Complete to permit all common uses of the object
- Primitive whose efficient implementation can only be achieved with access to the undelying data representation.

Classes

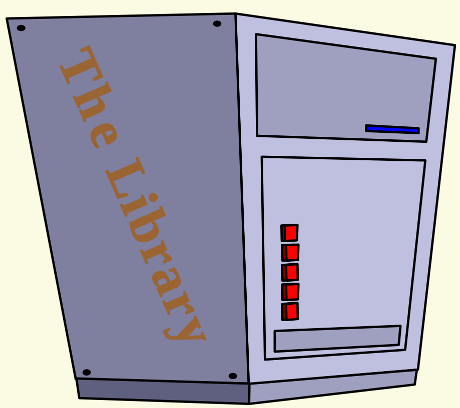
- A class has the following characteristics:
- represents real world object
- is a template to create objects (variables)
- encapsulates protected data with the operation on the data
- access to protected data is through calls (messages) that invoke operations
- In Ada a class may be a package that encapsulates a user defined data type and operations. There are many ways to create classes in Ada.

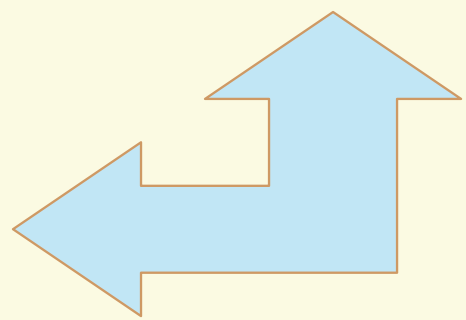
The Library



The compiler compiles the subunit checking all interfaces with library units. Resolves ambiguities. Places the subunit into the library.

The subunit is developed with calls to library units





The library maintains a record of all compiled subunits and their interfaces.

Context Clauses

- ✓ A context clause is the mechanism a programmer uses to introduce pre-compiled identifiers into the current compilation.

```
with Ada.Text_IO;                    -- The context clause
procedure HELLO is
  begin
         Ada.Text_IO.PUT_LINE("Hello, World!");
end HELLO;
```

Context Clause

- ✓ In the previous example, the call to Ada.Text_IO.PUT_LINE is more than just a subroutine call
- ✓ The compiler checks the call to make sure that the parameter signature (number, order, and type of parameters) is correct
- ✓ For this reason, all called units must either be pre-defined library units or units that you have already compiled

Visibility

- ✓ means you can programmatically have access to the item (subunit, type, and/or object).
- ✓ Need to follow the rules in section 8.3 of the ARM

Basics of the language

- ✓ Once you understand strong typing and Ada’s library, the rest of the language resembles most other high-order programming language (except for tasking - covered later)
- ✓ Following are some slides which highlight Ada-specific language features

Ada Reserved Words

abort	declare	function	of	renames
abs	delay	generic	or	return
accept	delta	goto	others	reverse
access	digits	if	out	select
all	do	in	package	separate
and	else	is	pragma	subtype
array	elsif	limited	private	task
at	end	loop	procedure	terminate
begin	entry	mod	raise	then
body	exception	new	range	type
case	exit	not	record	use
constant	for	null	rem	when
abstract	aliased	protected	while	with
requeue	tagged	until		xor

Identifiers

- Used as the names of various items (cannot use reserved words).
- Arbitrary length.
- First character - letter A - Z. *(ISO 10646-1)*
- Other characters - letter A-Z, numeral, _ *as a non-terminal character*, (but not several _ in a row).
- Not case sensitive.
- Comments are started by “--”, and continue to end of line

Examples of Identifiers

Legal Identifiers:

Reports Last_Name Window
DTZ Target_Location Pi
List Max_Refuel_Range

Illegal Identifiers:

_BOMBERS 1X P-i
Target/Location CHECKS_
Last Name First__Name

Ada Statements (Verbs)

SEQUENTIAL

:=

null

procedure call

return

declare

CONDITIONAL

if

then

elsif

else

case

TASKING

delay

entry call

abort

accept

select

requeue

protected

ITERATIVE

loop

exit

for

while

OTHER

raise

goto

All block constructs terminate with an *end* (end if, end loop, …)

Assignment Statements

VARIABLE := EXPRESSION ;

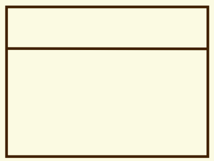
* The variable takes on the value of the expression

* The variable and the expression must be of the same type

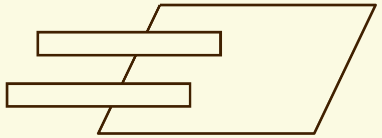
```
MY_INT := 17;                  -- Integer

LIST(2..4) :=LIST (7..9);      -- slice

TODAY := (13,JUN,1990);        -- aggregate

X := SQRT (Y);                 -- function call
```

Program Units

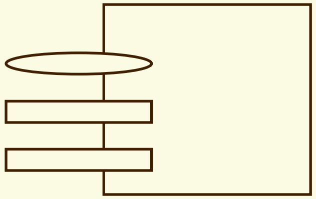
Subprograms
- -- Functions and Procedures
- -- Main Program
- -- Abstract Operations



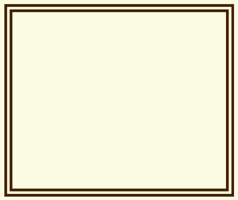
Tasks
- -- Parallel Processing
- -- Real-time Programming
- -- Interrupt Handling



Packages
- -- Encapsulation
- -- Information Hiding
- -- Abstract Data Types



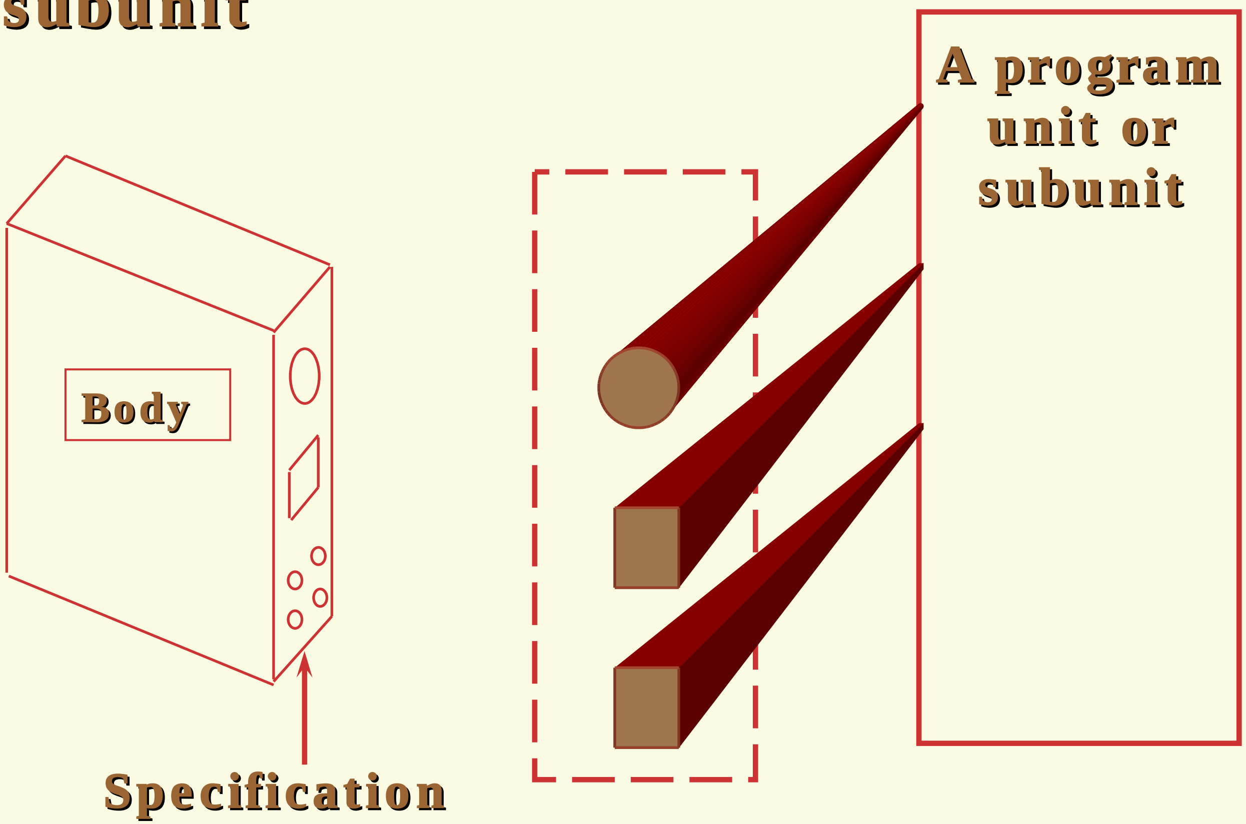
Generics
- -- Templates



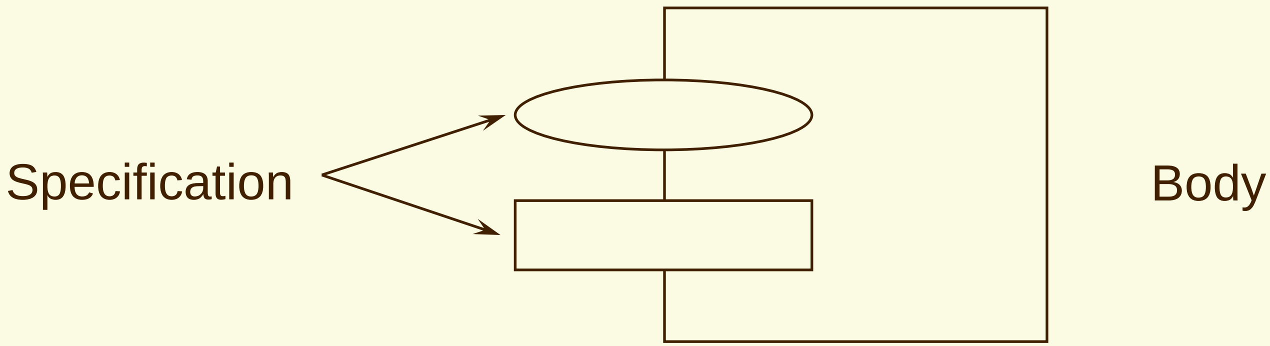
Specification and Bodies

"with" ing the package

Ada subunit



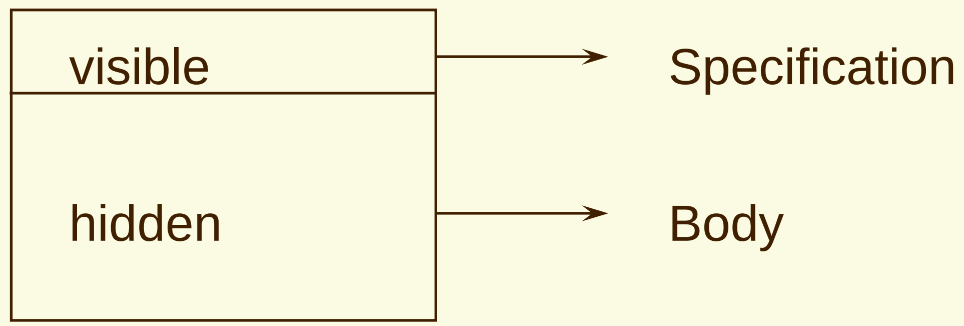
Subprogram Units



"What" the program unit does ← **ABSTRACTION** → "How" the program unit does what it does

All the user of the program unit needs to know ← **INFORMATION HIDING** → The details of implementation are inaccessible to the user

Ada Subprograms



- ✓ Procedures
  - Perform some "sub-actions“
  - Call always appears as a statement
- ✓ Functions
  - Calculate and return a value
  - Call always appears in an expression

Ada Procedures

```
-- Procedure Specification
procedure SWAP (PRE, POST: in out INTEGER);
-- Procedure Body
procedure SWAP (PRE, POST: in out INTEGER) is
      TEMP: INTEGER := PRE;
begin
      PRE    := POST;
      POST  := TEMP;
end SWAP;

-- Procedure Call
SWAP (MY_COUNT, YOUR_COUNT);
SWAP (MY_COUNT, POST => YOUR_COUNT);
SWAP  (PRE => MY_COUNT, POST =>  YOUR_COUNT);
```

Specification vs. body

-- Procedure Specification

```
procedure SWAP (PRE, POST: in out INTEGER);
```

- ✓ Required to allow calling of procedure SWAP prior to writing the actual body
- ✓ The specification allows the compiler to check the interface for units that call SWAP
- ✓ This is a COMPILABLE unit!

Specification vs. body

```
-- Procedure Body
procedure SWAP (PRE, POST: in out INTEGER) is
    TEMP: INTEGER := PRE;
begin
    PRE := POST;
    POST          := TEMP;
end SWAP;
```

- ✓ This is required before the compiled unit can actually run
- ✓ NOTE: if you are not making SWAP a library unit (i.e. if it is embedded inside another unit) the separate specification is not required. A body always implies a specification.

Ada Functions

-- Function Specification

```
function SQRT (ARG:  FLOAT) return FLOAT;
```

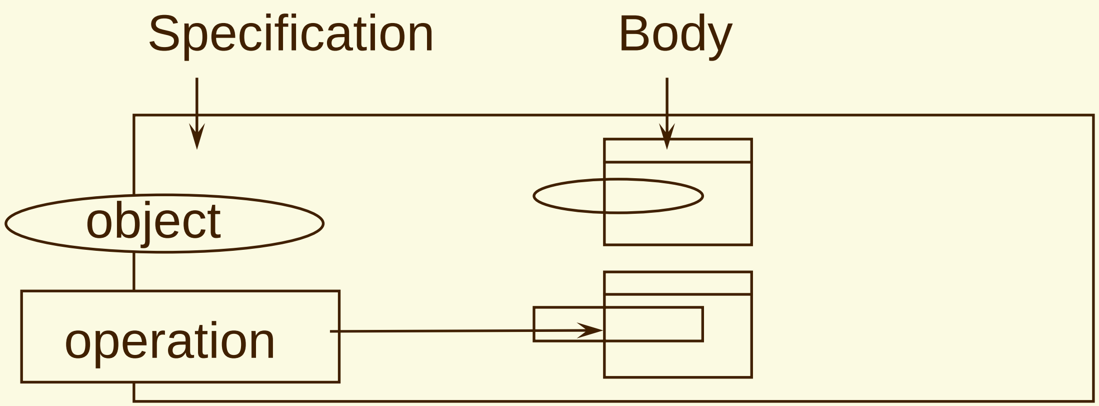
-- Function Body

```
function SQRT (ARG:  FLOAT) return FLOAT is
     RESULT:  FLOAT;
begin
       -- algorithm for computing RESULT goes here
     RETURN RESULT;
end SQRT;
```

-- Function Call (Assumes STANDARD_DEV and VARIANCE are of type FLOAT)

```
STANDARD_DEV := SQRT (VARIANCE);
```

Ada Packages



* The PACKAGE is the primary means of "extending" the Ada language.

* The PACKAGE hides information in the body thereby enforcing the abstraction represented by the specification.

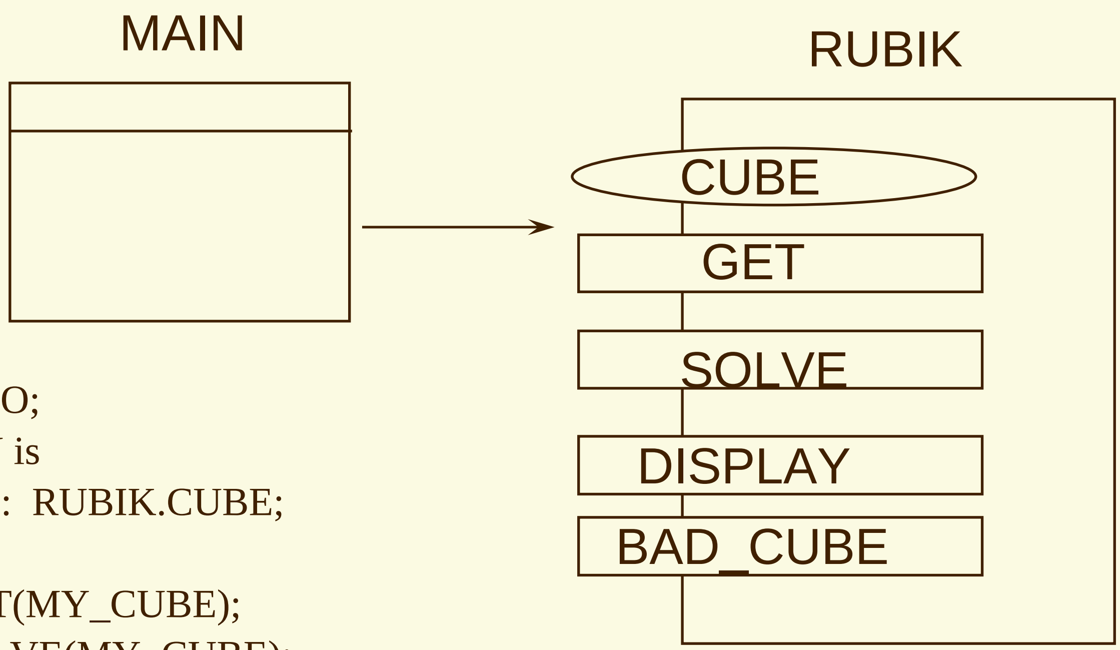
* Operations (subprograms, functions, etc) whose specification appear in the package specification must have their body appear in the package body.

* Other units (subprograms, functions, packages, etc) as well as other types, objects etc may also appear in the package body. If so, they are not visible outside the package body.

Ada Packages

```
-- Package Specification
package RUBIK is
        type CUBE is private;
        procedure GET (C : out CUBE);
        procedure SOLVE (C : in out CUBE);
        procedure DISPLAY (C : in CUBE);
        BAD_CUBE : exception;
private
        type CUBE is . . .
end RUBIK;

-- Package Body
package body RUBIK is
        -- all bodies of subprograms found in the
        -- package spec go here along with any
        -- other local declarations that should
        -- be kept "hidden" from the user
        procedure GET (C : out CUBE) is . . .
        procedure SOLVE (C : in out CUBE) is . . .
        procedure DISPLAY (C : in CUBE) is . . .
end RUBIK;
```

Package Usage




```
with RUBIK;
With Ada.Text_IO;
procedure MAIN is
     MY_CUBE :  RUBIK.CUBE;
begin
     RUBIK.GET(MY_CUBE);
     RUBIK.SOLVE(MY_CUBE);
     RUBIK.DISPLAY(MY_CUBE);
exception
     when RUBIK.BAD_CUBE =>
               Ada.Text_IO.PUT_LINE("You've got a bad one");
end MAIN;
```

Example (direct visibility)

```
Package MEASURES is
      type AREA is private;
      type LENGTH is private;
      function "+" (LEFT, RIGHT : LENGTH) return LENGTH;
      function "*" (LEFT, RIGHT : LENGTH) return AREA;
private
      type LENGTH is range 0..100;
      type AREA is range 0..10000;
end MEASURES;
--------------------------------------------------------------------------------
with MEASURES;  use MEASURES;  --direct visibility
procedure MEASUREMENT is
      SIDE1,SIDE2  :  LENGTH;
      FIELD :  AREA;
begin
      …...
      FIELD := SIDE1 * SIDE2;  --NOTE:  Infix notation of user-
                                      -- defined operation
end MEASUREMENT;
```

Example (indirect visibility)

```
Package MEASURES is
      type AREA is private;
      type LENGTH is private;
      function "+" (LEFT, RIGHT : LENGTH) return LENGTH;
      function "*" (LEFT, RIGHT : LENGTH) return AREA;
private
      type LENGTH is range 0..100;
      type AREA is range 0..10000;
end MEASURES;
-------------------------------------------------------------------------------
with MEASURES; --NOTE - no "use" clause
procedure MEASUREMENT is
      SIDE1,SIDE2  :  MEASURES.LENGTH;
      FIELD :  MEASURES.AREA;
begin
      …...
      FIELD := MEASURES."*"(SIDE1, SIDE2);
end MEASUREMENT;
```

Example (Ada95 compromise)

```
Package MEASURES is
      type AREA is private;
      type LENGTH is private;
      function "+" (LEFT, RIGHT : LENGTH) return LENGTH;
      function "*" (LEFT, RIGHT : LENGTH) return AREA;
private
      type LENGTH is range 0..100;
      type AREA is range 0..10000;
end MEASURES;
-------------------------------------------------------------------------------
with MEASURES;  -- NOTE:  no "use" clause
procedure MEASUREMENT is
use MEASURES.Length;    --direct visibility of type
use MEASURES.Area;      --direct visibility of type
      SIDE1,SIDE2  :  LENGTH;
      FIELD :  AREA;
begin
      …...
      FIELD := SIDE1 * SIDE2;
end MEASUREMENT;
```

Example (The best way)

```
Package MEASURES is
      type AREA is private;
      type LENGTH is private;
      function Add_Length (LEFT, RIGHT : LENGTH) return LENGTH;
      function Calc_Area (LEFT, RIGHT : LENGTH) return AREA;
private
      type LENGTH is range 0..100;
      type AREA is range 0..10000;
end MEASURES;
--------------------------------------------------------------------------------
with MEASURES;  -- NOTE:  no "use" clause
procedure MEASUREMENT is

      SIDE1,SIDE2  :  MEASURES.LENGTH;
      FIELD :  MEASURES.AREA;
begin
      …...
      FIELD := Measures.Calc_Area (Side1, Side2);
end MEASUREMENT;
```

Predefined Types

- Boolean
- Integer
- Natural (Subtype)
- Positive (Subtype)
- Mod (Modulus) --New
- Float
- Character
- Wide_Character --New
- String
- Wide_String --New
- Fixed
  Decimal_Fixed_Point --New

Ada 95 LRM
Core

- ✓ Section 1 through 13
- ✓ Annex A, “Predefined Language Environment”
- ✓ Annex B, “Interface to Other Languages”
- ✓ Annex J, “Obsolescent Features”

All implementations shall conform to the core language. In addition, an implementation may conform separately to one or more Specialized Needs Annexes.

Ada 95 LRM
Specialized Needs Annexes

- ✓ Annex C, "Systems Programming" *
- ✓ Annex D, "Real-Time Systems" *
- ✓ Annex E, "Distributed Systems"
- ✓ Annex F, "Information Systems"
- ✓ Annex G, "Numerics"
- ✓ Annex H, "Safety and Security"

Ada 95 LRM
Informative

- Annex K, “Language-Defined Attributes”
- Annex L, “Language-Defined Pragmas”
- Annex M, “Implementation-Defined Characteristics”
- Annex N, “Glossary”
- Annex P, “Syntax Summary”

Annexes

Three Annexes are required:

- Annex A, ``Predefined Language Environment''
- Annex B, ``Interface to Other Languages''
- Annex J, ``Obsolescent Features''

The following Specialized Needs Annexes define optional additions to the language.   A compiler including them, however, must be in full compliance.

- Annex C, ``Systems Programming''
- Annex D, ``Real-Time Systems''
- Annex E, ``Distributed Systems''
- Annex F, ``Information Systems''
- Annex G, ``Numerics''
- Annex H, ``Safety and Security''

Annex A

A.1 Package Standard

A.2 Package Ada --Parent

A.3.1 Package Ada.Characters

A.3.2 Package Ada.Characters.Handling

A.3.3 Package Characters.Latin_1

A.4 String Handling

A.4.2 String Maps

A.4.3 Fixed-Length String Handling

A.4.4 Bounded-Length String Handling

A.4.5 Unbounded-Length String Handling

A.4.7 Wide_String Handling

A.5 Numerics Packages *Pi & e 50*

A.5.1 Elementary Functions

A.5.2 Random Number Generator

A.10.8 Ada.Integer_Text_Io

A.10.9 Ada.Float_Text_Io

A.12 Stream Input-Output

A.15 Package Command_line

“If” Statement

Format

```
<if_statement> ::=
if <condition> then
   <sequence_of_statements>
{elsif <condition> then
   <sequence_of_statements>}
|else
   <sequence_of_statements>|
end if;
```

Conditional Expressions

- Expression where results are True or False.
- Qualifiers
  - =   /=   >   <   >=   <=   in   not in
- Boolean Operators
  - and   or   xor   not

Examples

```
type DAY is (Monday, Tuesday, ~~, Sunday);

    TODAY,  TOMORROW : DAY;

if TODAY = Sunday then              -- simple if-then
    TOMORROW := Monday;
end if;

if TODAY = Sunday then              -- if-then-else
    TOMORROW:= Monday;
else
    TOMORROW := DAY ' SUCC(TODAY);
end if;
```

Nested IF Example

```
type DIRECTION is (Left, Right, Back);
   ORDER :  DIRECTION;
if  ORDER = Left  then
    TURN_LEFT;
else
    if  ORDER = Right then
            TURN_RIGHT;
    else
            if  ORDER = Back then
                    TURN_BACK;
      end if;
    end if;
end if;
```

ELSIF Construct

Reduces level of nesting in a program

Example:

```
If ORDER = Left then
    TURN_LEFT;
elsif ORDER = Right then
    TURN_RIGHT;
elsif ORDER = Back then
    TURN_BACK;
end if;
```

More IF Examples

```
if  TODAY = FRIDAY then
    Num_Weeks := Num_Weeks + 1;
    Num_Left := Num_Left - 1;
    Ada.Text_IO.Put_Line("TGIF!!!");
end if;
 if     SCORE >= 90 then LETTER_GRADE := 'A';
   elsif SCORE >= 80 then LETTER_GRADE := 'B';
   elsif SCORE >= 70 then LETTER_GRADE := 'C';
   elsif SCORE >= 60 then LETTER_GRADE := 'D';
   else LETTER_GRADE := 'F';
   end if;
```

Short Circuit Control Forms

```
if COUNTER < TABLE'LAST then
        if RESULT(COUNTER) /= 0 then
                --          some statements
        end if;
end if;

if COUNTER < TABLE'LAST and then
        RESULT(COUNTER) /= 0 then
        --          some statements
end if;
if MY_PTR = null or else MY_PTR.ALL.ITEM = 0 then
        --          some statements
end if;
```

Sometimes, we run into the case of trying to use an if statement with an array, but the index might not be valid

Use the short circuit control forms to guarantee safe checks:
(and then, or else)

Why Avoid “not”

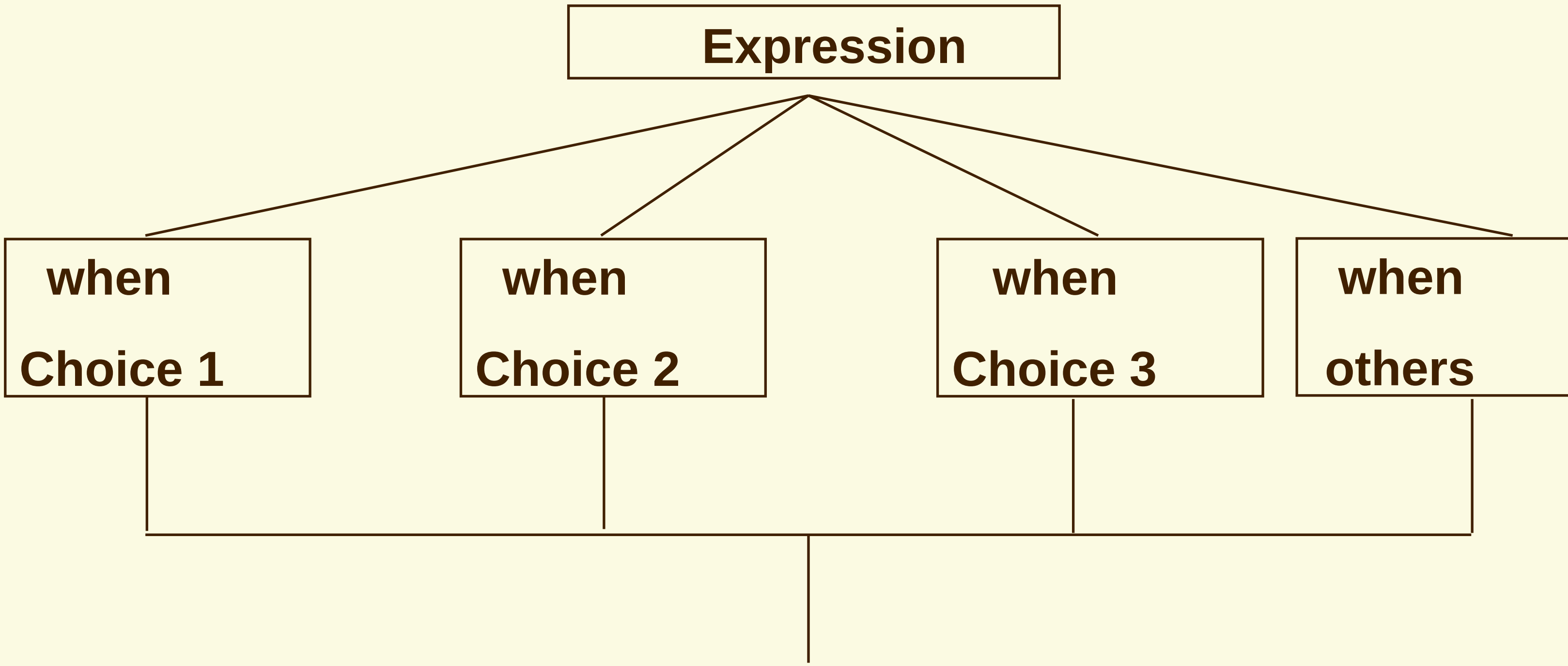
- ✓ “Not” can add unnecessary complexity
- ✓ you should use statements that there are no questions what is meant by a false out come

Case Statement

Format

```
case <discrete_expression> is
   when <choice> {|<choice>} =>
        <sequence_of_statements>
   {when <choice> {|<choice>} =>
        <sequence_of_statements>}
end case;
```

Expression



<discrete_expression>|

<discrete_range>|

others

“others”

- ✓ The rest (or all) receive this action or value
- ✓ Used in case statements, setting values, and exception handling
- ✓ Must be last alternative

Rules for ‘case’

- ✓ Expression must be discrete
- ✓ Choices must be mutually exclusive and exhaustive
- ✓ When "others" is used, it is only allowed as the last alternative

CASE Statement Examples

List of alternatives must be:

- discrete
- exhaustive, and
- mutually exclusive.

```
type Grade_Type is ('A','B','C','D','F','I');

case LETTER_GRADE is
   when 'A' => -- a sequence of statements
   when 'B' => -- a sequence of statements
   when 'C' => -- a sequence of statements
   when 'D' => -- a sequence of statements
   when 'F' => -- a sequence of statements
   when 'I'  => -- a sequence of statements
end case;

case LETTER_GRADE is
   when 'A' | 'B' | 'C' => -- a sequence of statements
   when 'D' =>             -- a sequence of statements
   when 'F' =>             -- a sequence of statements
   when others =>          -- a sequence of statements
end case;

case LETTER_GRADE is
   when 'A' .. 'C' => -- a sequence of statements
   when 'D' =>        -- a sequence of statements
   when 'F' =>        -- a sequence of statements
   when others =>     -- a sequence of statements
end case;
```

More CASE Examples

```
procedure SWITCH (HEADING :in out DIRECTION) is
begin
   case HEADING is
      when NORTH => HEADING := SOUTH;
      when EAST  => HEADING := WEST;
      when SOUTH => HEADING := NORTH;
      when WEST  => HEADING := EAST;
  end case;
end SWITCH;
```

```
case NUMBER is
   when 2          => DO_SOMETHING;
   when 3 | 7 | 8 => DO_SOMETHING_ELSE;
   when 9 .. 20   => DO_SOMETHING_RADICAL;
   when others    => DO_OTHER_THINGS;
                     Ada.Text_IO.PUT_LINE ("Others");
end case;
```

Better than “if ” ???

```
procedure SWITCH (HEADING :in out DIRECTION) is
 begin
   case HEADING is
     when NORTH => HEADING := SOUTH;
     when EAST  => HEADING := WEST;
     when SOUTH => HEADING := NORTH;
     when WEST  => HEADING := EAST;
  end case;
end SWITCH;
```

```
procedure SWITCH (HEADING :in out DIRECTION) is
 begin
    if Heading = North then
         Heading := South;
    elsif Heading = East then
         Heading := West;
    elsif Heading = South then
         Heading := North;
    else
         Heading := East;
    end if;
end SWITCH;
```

Summary of Case Statements

- ✓ Expression must be discrete
- ✓ Choices must be mutually exclusive and exhaustive
- ✓ When "others" is used, it is only allowed as the last alternative

ITERATIVE

STATEMENTS

Iterative Statements (Loops)

- ✓ Basic Loop -- used to execute a sequence of statements up to an infinite number of times.
- ✓ For .. loop -- used to execute a sequence of statements a finite number of times.
- ✓ While .. loop -- used to execute a sequence of statements until a certain condition is met.

Iterative Statement Examples

```
loop
   <sequence of statements>
end loop;

for I in 1 .. 10 loop
   <sequence_of_statements>
end loop;

while I < 10 loop
   <sequence_of_statements>
end loop;
```

Basic Loop

Basic loop is infinite because in embedded systems sometimes we want to loop continuously.

```
loop
        <sequence of statements>
end loop;
```

Basic Loop

EXAMPLES:

```
loop
    GET (SAMPLE);
    PROCESS (SAMPLE);
end loop;

loop
    GET (ALTITUDE);
    if ALTITUDE < 1000 and not GEAR_DOWN then
            LIGHT_GEAR_DOWN_LAMP;
            SOUND_ALARM;
    end if;
end loop;
```

Exit Statement

Two Forms:

Unconditional exit

```
exit;
```

Conditional exit

```
exit when <boolean expression>;
```

Basic Loop
‘exit’ Statement

```
loop
   ...
   if x = 20 then
      exit;
   end if;
end loop;

loop
   ...
   exit when x = 20;
end loop;
```

Basic Loop

EXAMPLES:

```
INDEX : integer := 1;
E        : Float    := 0.0;
TERM  : Float    := 1.0;
loop
        exit when INDEX := MAX_TERM;
        INDEX := INDEX + 1;
        TERM := TERM / FLOAT(INDEX);
        E :=  E + TERM;
end loop;
loop
        if INDEX = MAX_TERM then
                exit;
        else
              INDEX := INDEX + 1;
              TERM := TERM / FLOAT(INDEX);
              E := E + TERM;
        end if;
end loop;
```

Named Loops

```
OUTER:
loop
    INNER:
    loop
        exit OUTER;
    end loop INNER;
    exit;
end loop OUTER;
```

Exit Statement

exit; -- exits the innermost enclosing loop

exit Outer_loop; -- exits a named loop

OUTER_LOOP:

MIDDLE_LOOP:

INNER_LOOP:

exit;

exit OUTER_LOOP:

exit when INDEX = MAX_TERM;

Basic Loop

EXAMPLE:

```
X := ...
OUTER:
loop
    INNER:
    loop
        if X = 20 then
          exit OUTER;
        end if;
        exit INNER when X = 21;
        X := X + 2;
    end loop INNER;
end loop OUTER;
```

Exit Statement

exit when INDEX = MAX_TERM;

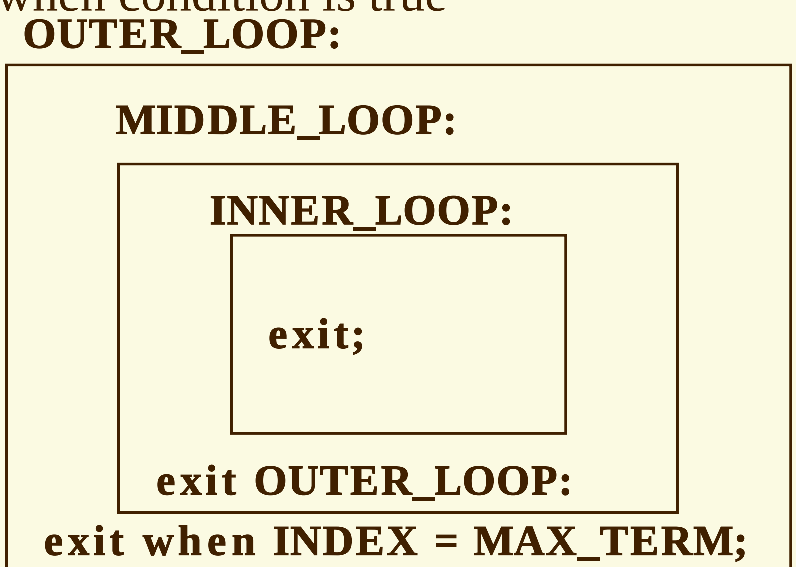
-- exits innermost enclosing

-- loop when condition is true

exit OUTER_LOOP when INDEX = MAX_TERM

exits the named loop

when condition is true



To go from the INNER_LOOP to the OUTER_LOOP we could use exit MIDDLE_LOOP

For Loop

for <loop index> in <start value> .. <stop value> loop

<sequence of statements>

end loop;

<loop index> is an identifier. It is declared implicitly--it is NOT declared by the programmer. Its type depends on the type of <start value> and <stop value>. It is treated as a constant within the loop.

<start value> and <stop value> must be discrete expressions.

Control Variables

- ✓ ARE IMPLICITLY DECLARED
- ✓ MUST BE DISCRETE
- ✓ TAKE THEIR TYPE FROM THE DISCRETE RANGE
- ✓ ARE IN EXISTENCE ONLY UNTIL end loop
- ✓ CANNOT BE MODIFIED (LOCAL CONSTANT)
- ✓ ONLY SINGLE STEP INCREMENT (DECREMENT)
- ✓ CAN 'HIDE' A VARIABLE WITH SAME NAME

Example of Control Variables

- for DAY in DAYS loop
- ...
- end loop;

- for COUNTER in reverse 1..10 loop
- ...
- end loop;

For Loop

EXAMPLES:

```
for INDEX in 1..15 loop
   --     Index is type Universal INTEGER
   <SOS>
end loop;

for INDEX in reverse  1..15  loop
   --     Backwards
   <SOS>
end loop;
```

For Loop

EXAMPLE:

```
procedure MAIN is
    SUM  : INTEGER := 0;
    I      : INTEGER := 10;
begin
    for I in 1..5 loop
                SUM := I + MAIN.I + SUM;
    end loop;
end MAIN;
```

For Loop

```
procedure MAIN is
    type DAYS is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
    subtype WEEKDAYS is DAYS range Mon .. Fri;
begin
    for  TODAY  in Mon .. Sun loop
        <sequence of statements>
    end loop;
    for TODAY in DAYS loop
      <sequence of statements>
    end loop;
   for TODAY in DAYS range Mon .. Fri loop
        <sequence of statements>
   end loop;
   for TODAY in WEEKDAYS loop
      <sequence of statements>
    end loop;
end;
```

For Loop

EXAMPLE:

```
with Ada.Text_IO;
procedure PRINT_ALL_VALUES is
      type COLORS is (RED, WHITE, BLUE);
      package COLOR_IO is new
         Ada.Text_IO.ENUMERATION_IO (COLORS);
begin
      for INDEX in 1..5  loop
         null;
      end loop;
      for A_COLOR in COLORS loop
         COLOR_IO.PUT(A_COLOR);
         Ada.Text_IO.NEW_LINE;
      end loop;
end PRINT_ALL_VALUES;
```

For Loop

```
with Ada.Text_IO;
procedure MAIN_PROGRAM is
        type INDEX_TYPE is range 1 .. 100;
        type CTR_TYPE is INTEGER range -10 .. 10;
        package INDEX_IO is new Ada.Text_IO.INTEGER_IO(INDEX_TYPE);
        package COUNTER_IO is new Ada.Text_IO.INTEGER_IO(CTR_TYPE);
begin
        for INDEX in INDEX_TYPE'FIRST.. INDEX_TYPE'LAST loop
                Ada.Text_IO.PUT("The index number is ");
                INDEX_IO.PUT(INDEX);
                Ada.Text_IO.NEW_LINE;
        end loop;

        for COUNTER in  CTR_TYPE'RANGE loop
                Ada.Text_IO.PUT("The counter number is ");
                COUNTER_IO.PUT(CONTER);
                Ada.Text_IO.NEW_LINE;
        end loop;
end MAIN_PROGRAM;
```

For Loop Example

```
procedure MAIN is
     type INDEX is range 0 .. 200;
begin
     for I in reverse INDEX loop
        <sequence of statements>
     end loop;

     for I in reverse INDEX range 1 .. 50 loop
        <sequence of statements>
     end loop;s`s
end MAIN;
```

Iterative Statement (While Loop)

```
while <boolean expression> loop
    <sequence of statements>
end loop;

while not Ada.Text_IO.END_OF_FILE(INPUT_FILE)
    loop
    ARGU_IO.GET(INPUT_FILE, ARGUMENT);
    Ada.Text_IO.PUT(ARGUMENT);
end loop;

while INPUT < 0 or INPUT > 100 loop          :
    Ada.Text_IO.PUT_LINE("Bad Input - Try again");
end loop;
```

While Loop

EXAMPLES:

```
while NOT_DARK loop
    PLAY_TENNIS;
end loop;
TURN_ON_LIGHTS;
while COUNT <= 100 loop
    COUNT := COUNT+ 1;
    if EMERGENCY then
      exit;
    end if;
end loop;
```

While Loop Example

```
package INT_IO is new
    Ada.Text_IO.INTEGER_IO(INPUT_TYPE);
package ENUM_IO is new
    Ada.Text_IO.ENUMERATION_IO(DONE_TYPE);

while INPUT not in 1 .. 99 loop
     Ada.Text_IO.PUT_LINE("Bad Input - Try again.");
    INT_IO.GET(INPUT);
end loop;

while not DONE loop
    INT_IO.GET(INPUT);
    Ada.Text_IO.PUT("More input (true/false)?");
    ENUM_IO.GET(Done);
end loop;
```

Nested Example

```
with Ada.Text_IO;
procedure COUNT_SMALL_LETTERS is
        CURRENT_LINE : STRING(1 .. 100);
        LENGTH : NATURAL;
        NO_SMALL_LETTERS : NATURAL := 0;
begin
        for LINE_NUMBER in 1 .. 10 loop
                Ada.Text_IO.GET_LINE(CURRENT_LINE, LENGTH);
                for CHAR_NUMBER in 1 .. LENGTH loop
                        if  CURRENT_LINE(CHAR_NUMBER) in  'a' .. 'z'  then
                                NO_SMALL_LETTERS :=
                                   NO_SMALL_LETTERS + 1;
                        end if;
                end loop;

                Ada.Text_IO.PUT("There are ");
                Ada.Text_IO.PUT(NATURAL'IMAGE(NO_SMALL_LETTERS));
                Ada.Text_IO.PUT_LINE(" small letters");
        end loop;
end;
```

Summary

✓ LOOPING CONSTRUCTS

- **BASIC LOOP**
  - execute an infinite number of times
- **FOR LOOP**
  - execute a finite number of times
- **WHILE LOOP**
  - execute until a certain condition is met

Review Program Units

- Subprograms (Procedure, Function)
- Packages
- Compilation Units:
- Library Units, Sub-units
- Relationships and Dependencies

Packages

What goes in a Package?

- **In the package specification => Basic Declarative items**
  - Constant object declarations
  - Type declarations
  - Subtype declarations
  - Subprogram specifications
  - Exception declarations
  - Object declarations (not a good idea because it makes them global)
  - Task specifications
  - Renaming declarations
  - Number declarations
  - Package specifications
  - Generic declarations
  - Generic instantiations
  - Deferred constant declarations
- **In the package body => Any Valid Declaration**

Packages

- Uses of Packages
  - Define groups of logically related items
  - Structuring tool for complex software systems
  - Define logically related resources
  - Useful for reusable software components

Packages

✓ Example

```
package PAY is

   type Pay_Type is digits 2 range 0.00 .. 10_000.00;

   procedure GET_WAGE (WAGE : out PAY_TYPE);

   function Calculate_Gross (Hours : in Integer;
                             Rate  : in Pay_Type) return Pay_Type;

end PAY;
```

Packages

```
with Ada.Text_IO;
with INT_IO;
with PAY_IO;
with PAY;
procedure PAYROLL is
   HRS, LEN :  INTEGER;
   WAGE, GROSS  : PAY.PAY_TYPE;
   NAME : STRING(1 .. 80);
begin
   -- Get Name, Hours, and Wage
   loop
      Ada.Text_IO.NEW_LINE;
      Ada.Text_IO.PUT("Input Name or quit:");
      Ada.Text_IO.GET_LINE(NAME, LEN);
   exit when NAME(1..LEN) = "quit";
      Ada.Text_IO.PUT("  Input Hours:");
      INT_IO.GET(HRS);
      PAY.GET_WAGE(WAGE);
      GROSS := PAY.CALCULATE_GROSS(HRS, WAGE);
      Ada.Text_IO.PUT("Gross Pay:");
      PAY_IO.PUT(GROSS, 5, 2, 0);
   end loop;
end PAYROLL;
```

Packages

```
package body PAY is
    function OVERTIME (HRS, WAGE : in PAY_TYPE)
        return PAY_TYPE is
    begin
        return 0.5 * WAGE * (HRS - 40.0);
    end;
    function CALCULATE_GROSS (HOURS : in INTEGER;
                              RATE  : in PAY_TYPE) return PAY_TYPE is
        PAY : PAY_TYPE;
    begin
        PAY := RATE * PAY_TYPE(HOURS);
        if HOURS > 40 then
            PAY := PAY + OVER_TIME(PAY_TYPE(HOURS), RATE);
        end if;
        return PAY:
    end CALCULATE_GROSS;
    procedure GET_WAGE (WAGE : out PAY_TYPE) is
        NUM       : PAY_TYPE;
        LENGTH, I : INTEGER;
        TEMP      : STRING(1 .. 80);
    begin       -- Rest is same as before
    end GET_WAGE;
end PAY;
```

Packages

- ✓ Benefits Of Ada Packages
  - Reusable Software Components
    - The Library Concept
  - Software Manageability
    - Break Up Program Into Smaller Manageable Chunks
  - Design Enforcement
    - Implementation Details Can Be Coded Later

Packages

- Packages Directly Support
  - ABSTRACTION
    - Packages are divided into two parts:
      - Specification
      - Body
  - INFORMATION HIDING
    - Details in the package body are inaccessible to
    - the package user.
  - MODULARITY
  - LOCALIZATION
  - CONFIRMABILITY
    - Using tested packages, the software system is easier to test.

Packages

- ✓ Goals Supported
  - MODIFIABILITY
    - Bodies may be changed without affecting the specification.
    - Contracts are formed with user.
  - EFFICIENCY
    - Details are isolated.
  - UNDERSTANDABILITY
    - Important parts are clear.
  - RELIABILITY
    - Errors are localized.

Packages

SPECIFICATION	Body
All public symbols, constants, types, subprograms.	Implementation details for the subprograms.

Packages

Example

```
package ROBOT_CONTROL is
   type STATUS is (ON, OFF);
   type SPEED is RANGE 0 .. 50;

   procedure RESET_SYSTEM;

   procedure RETURN_STATUS(CURRENT_STATUS : out STATUS);

   procedure RETURN_SPEED (CURRENT_SPEED : out SPEED);

   procedure INCREASE_SPEED (NEW_SPEED : in SPEED);
end ROBOT_CONTROL;
```

Packages

```
package body ROBOT_CONTROL is
   type DEVICES is (EYES, SPEAKER, ARMS, DRIVE_MOTORS);

   procedure RESET_SYSTEM is
   begin
      . . .
   end RESET_SYSTEM;
   procedure RETURN_STATUS(CURRENT_STATUS : out STATUS) is
   begin
      . . .
   end RETURN_STATUS;
   procedure RETURN_SPEED(CURRENT_SPEED : out SPEED) is
   begin
      . . .
   end RETURN_SPEED;
   procedure INCREASE_SPEED(NEW_SPEED : in SPEED) is
   begin
      . . .
   end INCREASE_SPEED;
begin
   RESET_SYSTEM;
end ROBOT_CONTROL;
```

Packages

```
with ROBOT_CONTROL;
procedure ROBOT_DRIVER is
   ROBOT_SPEED : ROBOT_CONTROL.SPEED;
   ROBOT_STATUS : ROBOT_CONTROL.STATUS;
begin
   ROBOT_CONTROL.INCREASE_SPEED(30);
   loop
      ROBOT_CONTROL.RETURN_SPEED(ROBOT_SPEED);
      exit when ROBOT_SPEED = ROBOT_CONTROL.SPEED'LAST;
      ROBOT_CONTROL.INCREASE_SPEED(ROBOT_SPEED + 1);
   end loop;
   loop
      ROBOT_CONTROL.RETURN_STATUS(ROBOT_STATUS);
      exit when ROBOT_STATUS = ROBOT_CONTROL.OFF;
   end loop;
end ROBOT_DRIVER;
```

Packages

- A Package Consists Of
  - Specification Required.
    - Define items which can be used from the package.
    - Defines visible information.
  - Body (Separately compilable)
    - Define items local to the package not visible to the user.
    - Contains the bodies of the exportable subprograms defined in the package specification.
    - Contains optional code executed at package elaboration time.

Packages

- Package Specifications
  - Anything listed in the public part of the package specification is "exportable".
  - The user "imports" the package resources using context clauses.
  - The "with" clause gives the user visibility to the package resources via expanded names.
  - The "use" clause gives the user direct visibility to the package resources via simple names.
  - A package specification defines a contract with the user of the package.

- What I can do for you...

Package Bodies

– If a unit (subprogram, package, task, generic) specification occurs in the package specification then the unit body must occur in the package body.
– Packages serve as repositories of logically related entities.
– Packages are (almost) always passive.
– The optional sequence of statements in the package body is executed one time when the package is elaborated. In other words, the package body may contain initialization code.
– A package body can be separately compiled from the parent which contains the package specification.

Packages - an invalid method

```
package METRIC_CONVERSIONS is
    CM_PER_INCH      : constant := 2.54;
    CM_PER_FOOT      : constant := CM_PER_INCH * 12;
    CM_PER_YARD      : constant := CM_PER_FOOT * 3;
    KM_PER_MILE      : constant := 1.609_344;
end METRIC_CONVERSIONS;
```

You don't need a package body with this package specification since there are no subprograms declared. Under Ada95 a package body for this type of spec is illegal. You must have at least one subprogram.

Packages

Specification

```
package MATH_FUNCTIONS is

    type NUMBERS is digits 7 range -10.0 .. 1_000_000.0;
    subtype POSITIVE_NUMBERS is NUMBERS range
        0.0 .. 1_000_000.0;

    function SQRT(A_NUMBER : in POSITIVE_NUMBERS)
        return POSITIVE_NUMBERS;

    function LOG(N : in POSITIVE_NUMBERS) return NUMBERS;

end MATH_FUNCTIONS;
```

Packages

Driver

```
with MATH_FUNCTIONS;
procedure MAIN_DRIVER is
    LOCAL_NUMBER : MATH_FUNCTIONS.NUMBERS := 100.0;
begin
    LOCAL_NUMBER := MATH_FUNCTIONS.LOG
                          (LOCAL_NUMBER);
    LOCAL_NUMBER := MATH_FUNCTIONS.SQRT
                                (LOCAL_NUMBER);
end Main_Driver;
```

NOTE: the above can be compiled and entered in library, even though the body for MATH_FUNCTIONS isn't written yet.

Of course, it could not be executed

Visibility

A PACKAGE CAN BE MADE AVAILABLE IN TWO DISTINCT WAYS

-- It can be textually nested

-- It can be accessed from a library

```
package COMPLEX is
   type NUMBER is
      record
         REAL_PART : FLOAT;
         IMAGINARY_PART : FLOAT;
      end record;
   function "+"(X, Y : in NUMBER) return NUMBER;
   function "-"(X, Y : in NUMBER) return NUMBER;
   function "*"(X, Y : in NUMBER) return NUMBER;
end COMPLEX;
```

Nested vs Library Units

```
procedure MAIN is
   package COMPLEX is
      type NUMBER ...
      function "+"...
      function "-"...
      function "*"...
   end COMPLEX;
   ...
   package body COMPLEX is
  separate;
begin
   -- The resources in COMPLEX are
  visible here.
end MAIN;
```

Packages As Library Units

```
with COMPLEX;
procedure MAIN is
begin
   -- The resources in COMPLEX
 are visible here.
end MAIN;
```

Direct Visibility

- Achieved through the "use" clause.
- Allows for naming of declarative items without having to preface with the package name.
  - use <package_name> { ,<package_name> } ;
- Recommended for limited application only Better to “use” just a type. This allows “infix” operators, but limits
- the direct visibility to a single type.

Friends don't Let Friends use *Use*

- Leads to problems during maintenance
- Makes debugging difficult
- Pollutes the *name space*

Packages

✓ Use Clause

```
procedure MIXUP is
   package TRAFFIC is
      type COLOR is (RED, AMBER, GREEN);
   end TRAFFIC;
   package GRAPHICS_STUFF is
      type COLOR is (RED,GREEN,BLUE);
   end GRAPHICS_STUFF;
   use TRAFFIC;
      -- Color, Red, Amber, Green become directly visible.
   use GRAPHICS_STUFF;
      -- Two homographs for Red and Green,
      -- Blue becomes directly visible,
      -- BUT Color becomes hidden, because there could be two Colors visible!!
   SIGNAL : TRAFFIC.COLOR;
   PIXEL : GRAPHICS_STUFF.COLOR;

begin
   SIGNAL := GREEN;                  -- Referential Disambiguation
   PIXEL  := RED;                    -- Referential Disambiguation
   PIXEL  := BLUE;                   -- OK
end MIXUP;
```

Compilation Issues

- Library unit specifications and bodies are normally kept in separate source files with descriptive file names for each. This allows you to take maximum advantage of Ada's separate compilation facilities.
- A specification must be compiled before its body. The specification is what introduces the resources into the library for other library units to use.
- The dependencies created among library units through "with"-ing form a hierarchy with the least dependant units at the top and the most dependant units at the bottom.
- Any time the specification for a library unit changes, all library units dependant on it must be recompiled in the order of their dependence. This creates a ripple effect throughout the library. Automatic tools to manage library recompilations are a great asset.
- Changing the body of a library unit does not cause any other library unit to be recompiled.

Summary

- Packages are one of the major innovations in Ada.
- Packages allow for collecting logically related entities into the library for use by Ada programs.
- Packages are the mechanism for enforcing information hiding, allowing the programmer to create abstract data types.
- Dependencies created by "with"-ing packages from the library require careful management of compilation order which is best left to automated tools.

Subprograms

Subprograms

- A program unit whose execution is invoked by a subprogram call.
- Two forms:
  - Procedure
  - Function
- Basic unit for expressing an "action" abstraction.
- These are the atomic building blocks of Ada programs.
- Procedures and functions specify operations for use in a program or as part of a larger module (a package or another subprogram).

Subprograms

- ✓ Procedures - Perform an action based on some input values, alter the state of some variables, or return result values to the invoker.
- ✓ Functions - Perform an action based on some input values and return a result value to the invoker.
- ✓ Every subprogram should be treated as a 'black box' and therefore should not directly reference data declared outside of the subprogram.

Subprogram names

- ✓ Subprogram names
- ✓ Should describe what the procedure does.
- ✓ Should be action oriented.
- ✓ Should describe the abstraction performed.

PARAMETER MODE

- The direction (from the standpoint of the subprogram) in which the value associated with the formal parameter is passed
- Three modes
  - in - Parameter may only be read
  - out - Parameter may be updated
  - in out - Parameter may be both read and updated
- Functions may only have *in* parameters
- The default parameter mode is *in*

Parameter Modes

- Parameter requirements
  - in
    - The caller must supply a value which can be a literal object, a constant object, a variable object, or an expression.
  - in out
    - The caller must supply a variable that contains a value.
  - out
    - The caller must supply a variable. The variable does not need to already have a value, since it will receive a value from the subprogram.

Example

✓ Procedure example

```
procedure EXAMPLE  (FIRST : in CHARACTER;
                    SECOND : in out CHARACTER;
                    THIRD  : out    CHARACTER) is

   LOCAL_CHARACTER : CHARACTER := 'A';

begin

   LOCAL_CHARACTER := FIRST;  -- okay
   FIRST := LOCAL_CHARACTER;  -- illegal
   LOCAL_CHARACTER := SECOND;  -- okay
   SECOND := LOCAL_CHARACTER;  -- okay
   LOCAL_CHARACTER := THIRD;  -- illegal
   THIRD := LOCAL_CHARACTER;  -- okay
end EXAMPLE;
```

Subprograms

More examples

```
procedure ADD(FORMAL_LEFT : in INTEGER;
              FORMAL_RIGHT : in INTEGER;
              FORMAL_RESULT : out INTEGER) is
begin
   FORMAL_RESULT := FORMAL_LEFT + FORMAL_RIGHT;
end ADD;
```

Driver

```
with ADD;
procedure ADD_VALUES is
   ACTUAL_LEFT : INTEGER := 5;
   ACTUAL_RESULT : INTEGER;
begin
    ADD(ACTUAL_LEFT, 2, ACTUAL_RESULT);
end ADD_VALUES;
```

Subprograms

Parameter requirements for "in" mode

```
procedure PRT_CHAR(FORMAL_CHARACTER : in CHARACTER);
```

Driver

```
with PRT_CHAR;
procedure VALID_PARAMETERS is
   THE_CHARACTER_C : constant CHARACTER := 'C'
   CHARACTER_1 : CHARACTER := 'C'
begin
   PRT_CHAR('C');                      -- Literal object
   PRT_CHAR(THE_CHARACTER_C);          -- Constant object
   PRT_CHAR(CHARACTER_1);              -- Variable object
end VALID_PARAMETERS;
```

Subprograms

Parameter requirements for "in out" mode

```
procedure INCREMENT(FORMAL : in out CHARACTER);
```

Driver

```
with INCREMENT;
procedure VALID_PARAMETERS is
   THE_CHARACTER_A : constant CHARACTER := 'A';
   CHARACTER_1 : CHARACTER := 'A'
begin
   INCREMENT('A');                     -- illegal
   INCREMENT(THE_CHARACTER_A);   -- illegal
   INCREMENT(CHARACTER_1);          -- okay
end VALID_PARAMETER;
```

Subprograms

Parameter requirements for "out" mode

```
    procedure GET_NAME(FORMAL : out STRING);
```

Driver

```
    with GET_NAME;
    procedure VALID_PARAMETERS is
        NAME_PROMPT : constant STRING := "Name please?";
        PERSONS_NAME : STRING(1 .. 20);
    begin
        GET_NAME("Person's name");      -- illegal
        GET_NAME(NAME_PROMPT);       -- illegal
        GET_NAME(PERSONS_NAME);     -- okay
end VALID_PARAMETER;
```

Subprograms

Expressions as actual parameters

```
procedure PRINT_VALUE(FORMAL : in INTEGER);
```

Driver

```
with PRINT_VALUE;
procedure EXAMPLE is
   A_VALUE : INTEGER := 25;
begin
    PRINT_VALUE  (A_VALUE ** 2 * 25 + (3 * 2 - 5));
end EXAMPLE;
```

Subprograms

Parameter association

When the subprogram call is made, the actual parameters are associated with the formal parameters.

```
procedure ADD (LEFT     : in INTEGER;
               RIGHT    : in INTEGER;
               RESULT   : out INTEGER);
```

Positional association

```
ADD(5, 10, TEMP);
ADD(FIRST, SECOND, NEW_VALUE);
```

Named association

```
ADD(LEFT => 5, RIGHT => 10, RESULT => TEMP);
ADD(RESULT => TEMP, LEFT => 5, RIGHT => 10);
```

Mixing notations

```
ADD(5, 10, RESULT => TEMP);               -- okay
ADD(5, RESULT => TEMP, RIGHT => 10);      -- okay
ADD(LEFT => 5, 10, TEMP);                 -- Illegal
```

Subprograms

Default values

Can be given for "in" parameters only. Useful when there is a long parameter list, and some of the parameters lend themselves to default values.

Example

```
procedure PUT (ITEM   : in FLOAT;   FORE : in NATURAL := 0;
               AFT    : in NATURAL := 2;
               EXP    : in NATURAL := 0);
```

Driver

```
with PUT;
procedure EXAMPLE is
   FOO : FLOAT := 3.14;
begin
   PUT(FOO);
   PUT(ITEM => FOO; AFT => 4);
end EXAMPLE;
```

Functions

- ✓ Functions
  - A function is a high-level operator that returns a single value for use in an expression.
- ✓ Functions may only have "in" parameters.
- ✓ Functions must return a value using the "return" statement.

Functions

Function example

```
procedure MAIN is

   MY_NUMBER : FLOAT := 10.0;

  function INVERSE   (NUMBER : in FLOAT)
     return FLOAT is
  begin
     return 1.0 / NUMBER;
  end INVERSE;

begin
  MY_NUMBER := INVERSE(MY_NUMBER);
end MAIN;
```

Functions

Function example

```
function AVERAGE_OF  (FIRST,
                      SECOND : in TEST_SCORES)
   return TEST_SCORES is

begin
   return (FIRST + SECOND) / 2.0;
end AVERAGE_OF;
```

Example

Function example

```
with MY_TYPES;     use MY_TYPES;
function LESSER_OF  (FIRST, SECOND : in A_TYPE)
                                        return A_TYPE is
begin
   if FIRST < SECOND then
      return FIRST;
   else
      return SECOND;
   end if;
end LESSER_OF;
```

Driver

```
with MY_TYPES;     use MY_TYPES;
with LESSER_OF;
procedure MAIN is
     MINIMUM, NUMBER_1, NUMBER_2 : A_TYPE;
begin
   -- give NUMBER_1 and NUMBER_2 values

   MINIMUM := LESSER_OF(NUMBER_1, NUMBER_2);
end MAIN;
```

Subprograms

To exemplify some of the Ada statements, lets look at the implementation of a 'wrap-around successor function for type DAYS.

```
procedure TEST is
   type DAYS is (SUN, MON, TUE, WED, THU, FRI, SAT);
   TODAY, TOMORROW : DAYS;
   function WRAP(DAY : in DAYS) return DAYS is
     begin
            if Day = Days’last then
                        return Days’first;
            else
                        return Day’succ;
            end if;
     end Wrap;
begin
   TOMORROW := WRAP (TODAY);
end TEST;
```

Subprograms

The name of a function can be any identifier as we have seen or it can be one of the predefined operators. This is used to "overload" the meaning of the predefined operators.

```
with VECTORS;
use VECTORS;
procedure MAIN is
   function "*"(X, Y : in VECTOR) return FLOAT is separate;
          -- An infix operation on vectors
   A, B : VECTOR(1 .. 20);
   RESULT : FLOAT;
begin
   RESULT := A * B;   -- Use of the infix notation
--OR
     RESULT := "*"(A,B);
--OR
     RESULT := VECTORS."*"(A,B) --use not needed
end MAIN;
```

Subprograms

Name Overloading

Both procedure and function names can be overloaded. The compiler determines which one to invoke based on the types of the actual parameters used in the call.

```
function "*"(X, Y : in VECTOR)  return FLOAT;
function "*"(X, Y : in INTEGER) return INTEGER;
function "*"(X, Y : in FLOAT)   return FLOAT;
```

Note that the overloading operators can lead to debugging or maintenance problems. It might be better to have a procedure named MULTIPLE that takes in two vectors. This makes it obvious that an user-writer procedure is being called.

Overloading I/O

- ✓ Overloading predefined I/O operations, however, does not lead to such confusion.

```
procedure PUT(X : in INTEGER);
procedure PUT(X : in STRING);
procedure PUT(X : in FLOAT);
```

Summary

- ✓ Subprograms are the basic unit of functional decomposition.
- ✓ There are three parameter passing modes with strict enforcement on the uses of the parameters in the subprogram.

Summary (cont.)

- ✓ There is ample flexibility in parameter association in the call to the subprogram, including named association for improved readability.
- ✓ Subprogram names can be overloaded, and the context of the call is used to choose the appropriate code to execute. (This is done at compile time.)

Exercise

- Modify your previous program.
- Create a package called Age_Package. Have it contain the type and the read and print operations.
- "With" the package in a main program. Create a variable of type Age. Input a value, and then echo print. Note - you should not "with" any other package in your main program.
- Then modify your program to read from command line. Hint "Ada.Command_line"

Review Ada Types and Objects

- Predefined vs. User-Define
- Attributes
- Type Conversion
- Scalar, Access and Private Types
- Use of Appropriate Types

Scalar Data Types

TYPING ENFORCES:

- Abstraction, hiding of implementation details (simulates real world events)
  - Properties of objects and operations are separated from underlying and internal implementation - dependent properties
    - Object_A : Fruit; (Don't really care about the details of fruit)

TYPING ENFORCES:

✓ FACTORIZATION OF PROPERTIES, MAINTAINABILITY

- Common properties of objects are described and collected in one place
- A name is associated with that description
- Can change properties of objects by changing only the type declaration

Typing

- ✓ Typing is the enforcement of the class of an object.
- ✓ It prevents inadvertent conversion of one type to another.
- ✓ Very strong typing prevents the conversion of one type to another.
- ✓ Strong typing requires explicit action on the part of the implementor to “coerce” one type into another.
- ✓ Ada supports strong typing. It provides both automatic and used-defined coercion.

More on Typing

- Without type checking, a program can crash at run-time for mysterious reasons
- Typing allows early error detection
- Type declarations are part of design. Helps with documentation
- More efficient code can be generated.

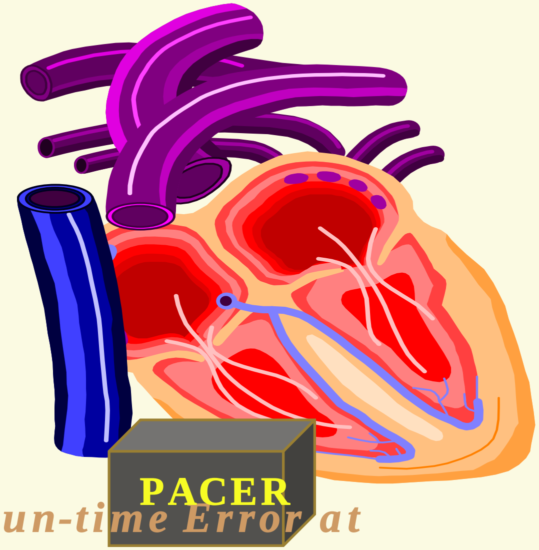
FORCES DESIGN DECISIONS TO BE MADE EARLY!!

Strong Typing Example

--Systolic -- heart pumping out 90 - 140

--Diastolic -- heart filling up 50 - 90

```
type Systolic is range  90 .. 140;

type Diastolic is range  50 .. 90;

Systolic_Init_Value : constant  Systolic := 110;
Diastolic_Init_Value : constant  Diastolic := 60;

Systolic_Reading   : Systolic := Systolic_Init_Value ;
Diastolic_Reading : Diastolic := Diastolic_Init_Value;
```

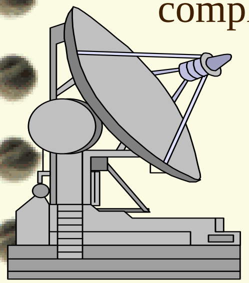



```
Systolic_Set_Value  : Systolic := 200; --Warning of run-time Error at compile
Diastolic_Set_Value : Diastolic;

Systolic_Set_Value := Systolic_Reading + Diastolic_Reading;  -- Error at compile
```

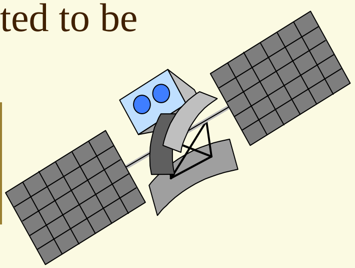
--type inconsistency in this expression

Problems of Weak Typing

- ✓ The TOPEX/Poseidon satellite entered into **<u>safe hold mode</u>** on Aug. 27, 1992 at approximately 11:13 a.m. PDT. This incident occurred during the inclination maneuver about 6 seconds from the end of the propulsion burn.
- ✓ The inclination maneuver was successful and placed the satellite in the **proper 66 degree inclination toward the Earth.**
- ✓ Project managers have determined that the safe hold mode was the result of a bug" in the software code which set the failure detection correction limit for a roll angle of 3 degrees, not 7 degrees as intended. This was a result of residual LANDSAT code which did not correlate to the **<u>program design language</u>**.
- ✓ All satellite hardware is functioning properly based on detailed review of the maneuver playback data. Re-configuration of the satellite back to the standard configuration prior to safe hold mode is in process at this time and is expected to be completed successfully.



```
type Inclination is range -7..7;
Inclination_Upper_Max : constant Inclination := 7
```



Types

- ✓ Characteristics
  – set of values
  – set of primitive operations

Type Definition

type Some_Type is (definition of what the type is);

Subtypes

- ✓ used to further limit values of abstraction
- ✓ has all primitive operations of base-type
  - subtype Sub_Type is Some_Type
    range Ec..Ee;

Objects

A run-time entity that contains a value of the object's type.

Object Definition

Object : Type_Name
[:=initial value];

Access Types

(Pointers)

Access Types

- ✓ Designates an object by an allocator
- ✓ Only object in Ada that has a default value (set to null if not initialized)
- ✓ must be defined with a data structure or subprogram type to pointed at.

Access Types

Access types point to data structures, subprograms, or other access types

```
type Name_Type is string (1..10);
Type Name_Ptr_Type is access Name_Type;
type Int_Ptr is access Integer;
type Vehicle_Ptr is access all Vehicle'class;
type Procedure_Ptr is access procedure;  --points to any
                                         --parameter less procedure
IP : Int_Ptr;
I : aliased Integer;        --aliased allows I to be pointed to
IP := I'Access;
IP.all := 42;               --I must be de-referenced as all
IP := new Integer'(I);      --makes new values, copies I into it
```

Problem with Pointers

- It is often convenient to declare a pointer to a data object and use this pointer as a parameter.

- The problem: even if you make this parameter a *read only* parameter via the *in* mode, the function/procedure can change what the pointer points to (rather than the pointer itself).

- To prevent this, there is a mechanism that makes a pointer and what it points to *read only*.

Constant Access Types

Replace the word *all* in the type definition by the word *constant*.

```
type Int_Ptr is access constant Integer;
```

Now, variables of type Int_Ptr can point to Integers, but what they point to may not be modified.

```
IP : Int_Ptr;
I : aliased Integer; --aliased allows I to be pointed to

IP := I'Access;

IP := new Integer' (I);         --legal, not modifying what IP points to
IP := new Integer' (5);         --new value, original still not modified

IP.all := 5;                    --illegal, compiler error.  IP is read only
```

This allows you to declare a pointer type, pass it as a parameter, and prevent the procedure/function from modifying what the pointer points to.

Dynamic Selection

An access type can refer to a subprogram; an access-to-subprogram value can be created by the **‘Access** attribute. A subprogram can be called using this pointer. This allows you to include a pointer to a routine inside of a record, or as a parameter. This is known as a ***callback***.

```
type Trig_Function is access function (F : Float) return Float;

T : Trig_Function;
X, Theta : Float;

T := Sin'Access;
X := T(Theta);          -- implicit dereferencing.  SHOULD NOT BE USED!
                        -- This looks like normal function call.

X := T.all (Theta); -- explicit dereferencing.  THIS IS PREFERRED.
```

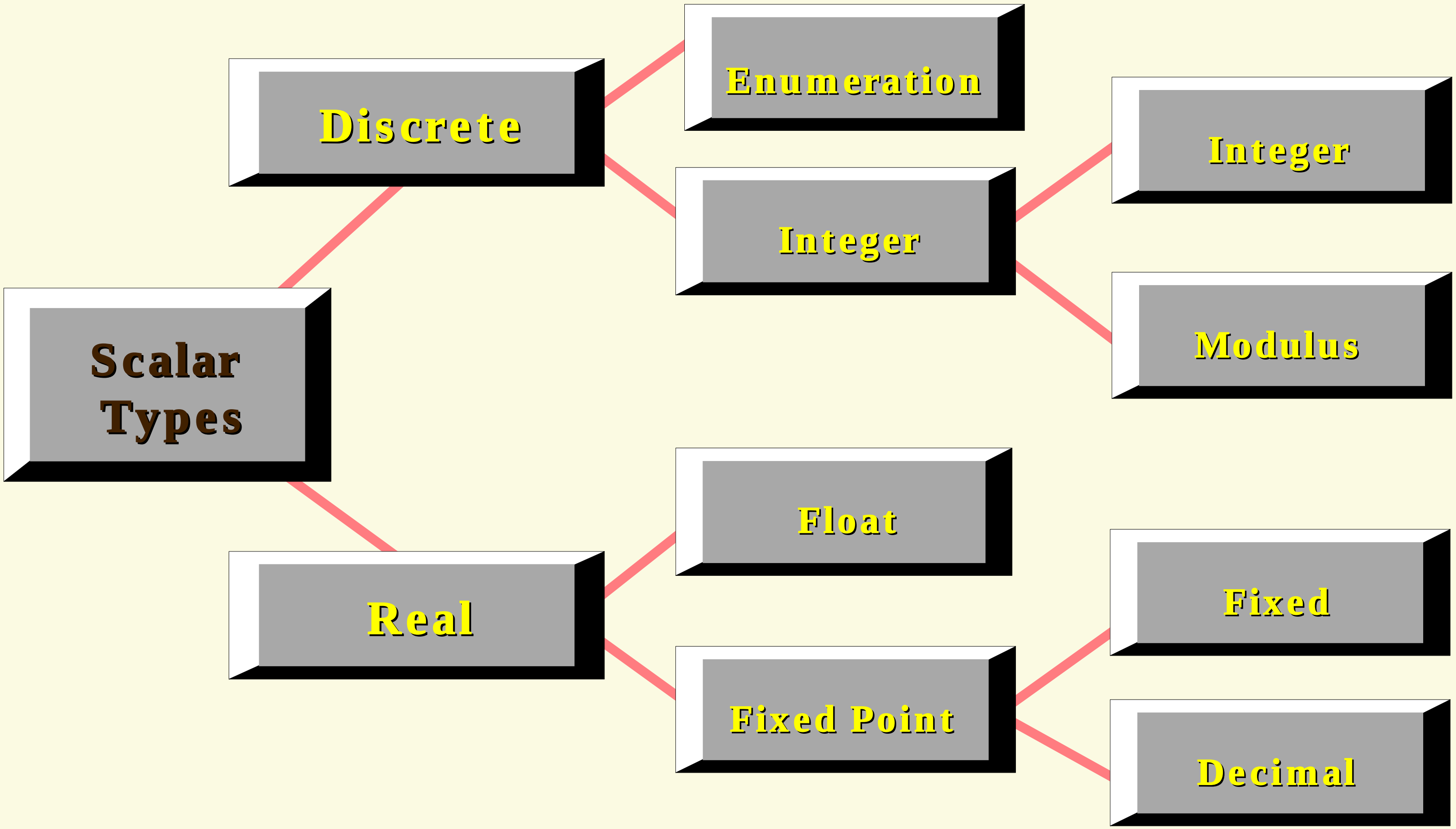
T can *point to* functions (such as Sin, Cos and Tan) that have a matching parameter list. Functions must have matching return types.

Classes of Ada Types

Ada Types

SCALAR	Objects are single values
COMPOSITE	Objects contain other components
PRIVATE	Objects are 'abstract'
ACCESS	Objects 'point' to other objects & subprograms
TASK	Objects are parallel processes
Protected	Coordinated access to shared data
Tagged	Inheritance & Run-time polymorphism

Scalar Types



Scalar Types
(One Object Elements)

- ✓ Discrete Types
  - Enumeration Types
  - Integer Types
    - Modulus
    - Integer
- ✓ Real Types
  - Float Types
  - Fixed Point Types
    - Fixed
    - Decimal

Discrete Types

A type defines only whole values (exact values).

Apple

1

A

Enumeration Types

- ✓ allows an abstraction to be represented directly
- ✓ can be used in indexing, iteration, case statements and record variants
- ✓ Use name of real world items

Enumeration Types

set of values: order set of distinct values

structure: (E1, E2, E3, … , En)

operations: assignment := membership in not in relation = /= < <= > >=

Defining Enumeration Types

```
type Week_Type is (Sun, Mon, Tue,
                   Wed, Thu, Fri, Sat);
type European_Week_Type is
      (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
subtype Work_Week_Type is Week_Type
   range Mon..Fri;
```

Predefined Enumeration Types

Boolean (False, True)

Character ISO 10646 256 code set (8 bits)

Latin 1 character set

Wide_Character ISO 10646 65536 code set (16 bits)

Attributes for Enumeration

format: S'Attribute [()]

First	Last	Range	base
Min	Max	Succ	Pred
Val	Pos	Value	
Image	Wide_ Image		
Width	Wide_Width		

Examples of Attributes

```
Week_Type'First          = Sun
Week_Type'Last           = Sat
Week_Type'Pos(Mon)       = 1
Week_Type'Val(0)         = Sun
Work_Week_Type'Range = Mon..Fri
    =
      Work_Week_type'First..Work_Week_Type'Last
```

Enumeration Objects

```
Day       : Week_Type  := Tue;
Tomorrow  : Week_Type  :=
            Week_Type'Succ(Day);
Yesterday : Week_Type  :=
            Week_Type'Pred(Day);
Work_Day  : Week_Type range
            Mon..Fri     := Mon;
Class_Day : Work_Week_Type;
```

Enumeration Objects

```
Bored   : Boolean     := True;
Yes     : Character   := ‘y’;
Bell    : Character   :=

        Ada.Character.Latin_1.BEL;
Upper   : Character range ‘A’..’Z’;
```

Sample Program

```
With Ada.Text_IO;
procedure Main_Driver is

type Week_Type is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
package Week_Type_IO is new
    Ada.Text_IO.Enumeration_IO(Week_Type);

Day : Week_Type;
                    -- Attributes
begin
    Day := Week_Type ‘First;            -- Is Mon
    Week_Type_IO.Put(Day);
    Day := Week_Type 'Succ (Thu);       -- Is Fri
    Day := Week_Type 'Succ (Sun);       -- Error
end Main_Driver;
```

Representation Specs

Address	Object’address
Alignment	Object’alignment
Size (Object)	Object’size
Size (type)	Type’size

internal code (numeric) representation
for Type use (E1 => 1,
E2 => 3, …En => 999);
--number must be greater than the previous entry

Integer Types

- ✓ consecutive numeric literals that do not have a radix points
- ✓ two form
  - – decimal
    - • 12000 12e3 12_000
  - – based
    - • 2#1110_0000# 16#e#e1

Integer Type

Set of Values:	set of consecutive numeric literials
Structure:	range L..U (System.Min_Int..System_Max is the greatest range)
Set of Operations:	assignment :=
	membership in not in
	relation = /= < <= > >=
	math + - * / mod rem
	unary + - abs
	exponentiating **

Integer Rem and Mod

A	B	A/B	A rem B	A mod B
10	5	2	0	0
11	5	2	1	1
12	5	2	2	2
13	5	2	3	3
14	5	2	4	4

A	B	A/B	A rem B	A mod B
-10	5	-2	0	0
-11	5	-2	-1	4
-12	5	-2	-2	3
-13	5	-2	-3	2
-14	5	-2	-4	1

A	B	A/B	A rem B	A mod B
10	-5	-2	0	0
11	-5	-2	1	-4
12	-5	-2	2	-3
13	-5	-2	3	-2
14	-5	-2	4	-1

A	B	A/B	A rem B	A mod B
-10	-5	2	0	0
-11	-5	2	-1	-1
-12	-5	2	-2	-2
-13	-5	2	-3	-3
-14	-5	2	-4	-4

Modulus Type

Set of Values:	set of consecutive numeric literials starting at 0
Structure:	mod U (0..System_Max *2 + 2 is the greatest range)
Set of Operations:	assignment := membership in not in relation = /= < <= > >= math + - * / mod rem exponentiating ** (Auto wrap)

Defining Integer Types

```
type Pages_Type is
        range 0..System.Max_Int;
subtype Accnt_Type is integer
        range -1000..100_000;
type Byte is mod 255; -- an unsigned byte
type Hash_Index is mod 97;
```

Predefined Integer Types

- Integers
  - Integer range (-2**15)+1..(2**15)-1
  - Natural range 0..Integer'Last (subtype)
  - Positive range 1..Integer'Last (subtype)
  - Long_Integer range (-2**31)+1..(2**31)-1
- Modulus
  - NONE

Attributes for Integers

format: S'Attribute [()]

First Last	Range	Base	
Min	Max	Succ	Pred
Val	Pos	Value	
Modulus	Image	Wide_ Image	
Width	Wide_Width		

INTEGER Objects

type Hour_Type is range 0 .. 12;

type Minute_Type is range 0 .. 59;

Integer.Object Declaration:

Hour : Hour_Type;

Minutes : Minute_Type : = 0;

Hour	Minutes
undef	***0***

INTEGER Attributes

```
With Ada.Text_Io;
Procedure Test is
Type Hours is range 0 .. 12;
Type Days is ( Mon, Tues, Wed, Thr, Fri, Sat, Sun );
-- My_Int : Integer := 0;
-- My_Hour : Hours := Hours’Last;

--Hours’First           – 0
--Hours’ Last           – 12
--Hours’Succ(10)        -- 11
--Hours’Succ(12)        -- Error
--Days’Val (3)          --Thr
--Hours’Val (3)         -- 3
--Hours’Image           -- “12”
--Days’Image (Mon)      -- “Mon”
Begin
    ada.text_io.put_line (Hours’Image (Hours’Val (2) ));
    ada.text_io.put_line (Days’Image (Days’Val (2) ));


My_Int : Hour_Type := Hour_Type'First;
```

Sample Program

```
procedure Main_Driver is
   type ALTITUDE is range 0 .. 100;
   type DEPTH is range -100 .. 0;
   type DISTANCE is range 0 .. 200;

   METERS : ALTITUDE := 10;
   FATHOMS : DEPTH   := -25;
   MILES : DISTANCE   := 50;

begin
    FATHOMS:= 10;                              -- error
    MILES       := MILES + 50;
    METERS  :=  METERS + FATHOMS   -- error
    MILES      :=  MILES + DISTANCE (Meters);
end Main_Driver;
```

EXPLICIT TYPE CONVERSION

Representation Specs

- ✓ Address — Object'address
- ✓ Alignment — Object'alignment
- ✓ Size (Object) — Object'size
- ✓ Size (type) — Type'size

```
for Some_Name use expression;
for Some_Name use Name;
```

Rep Specs Examples

```
type Medium is range 0..65_000;
for Medium use 2*Byte;
Device_Register : Medium;
for Device_Register use Medium'Size;
for Device_Register use
   System.Storage_Elements.To_Address
   (16#FFFF_0020#);
```

Discreet Summary

- ✓ an ordered set of distinct values
- ✓ a Primitive set of operations
- ✓ a set of attributes to enhance understandability

Real Types

- ✓ Ada real numbers give only approximate representation of quantities.
  - A real type defines a set of model numbers that can be represented exactly.
  - The accuracy of predefined real types will vary among implementations.

Real Types

- For portability and for sake of abstraction, Ada allows you to define the error bounds of real types:
  - relative accuracy - float e.g. space systems
  - absolute accuracy - fixed e.g. voltage measuring equipment, Money

Real Types

- consecutive numeric literals that have a radix points
- two form
  - decimal
    - 98.6 9.86e1 0.986e2
  - based
    - 2#1010_1101.1010# 16#A.DA#e1

Floating Point

- ✓ an approximation of real numbers based on the number of digits
- ✓ accuracy is at least the precision of the number of decimal digits representable by objects of that type.

```
type Some_Name is digits X [range L..U];
```

Float Type

Set of Values:	set of approximations of real Numbers
Structure:	digits N [range L.X..U.X]
Set of Operations:	assignment :=
	membership in not in
	relation = /= < <= > >=
	math + - * /
	unary + - abs
	exponentiating **
	Math Functions (see A.5.1)
	Random Number (see A.5.2

Attributes for Floats

Address	Base	Ceiling	Compose
Copy_Sign	Denorm	Digits	First
Exponent	Floor	Fraction	Image
Leading_Part	Last	Model	Max
Machine_Emax	Machine_Emin		Machine
Machine_Radix	Machine_Overflow		Min
Machine_Rounds	Machine_Mantissa		Pred
Model_Emin	Model_Epsilon		Model_Small
Remainder	Rounding		Safe_First
Safe_last	Scaling	Size	Succ
Signed_Zero	Truncation	Unbiased_Rounding	
Valid	Wide_Image	Wide_Value	Value
Wide_Width	Width	Adjacent	

Floating Point Type Attributes

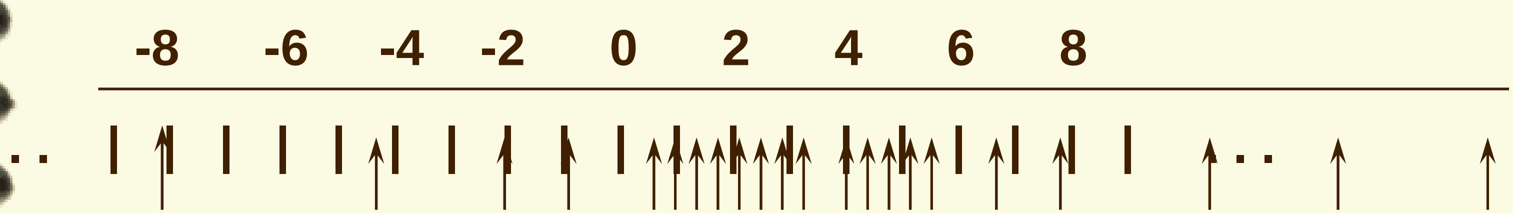
T ' Digits -- # of decimal digits in mantissa
T ' Mantissa
T ' Epsilon
T ' Emax = 4 * T ' Mantissa
T ' Model_Small = 2.0 ** (-T ' Emax-1)
-- smallest possible model #
T ' ???   = 2.0 ** T ' EMAX * (10 - 20** (-T '   MANTISSA))
-- largest possible model #
T ' Safe_First yield lower bound of safe # range
T ' Safe_Last   yield upper bound of safe # range

– Useful to determine the properties of the type (i.e. the computer being used)

Defining Floating Point

✓ Specify number of significant digits

```
type percentage is digits 4;
type Gpa is digits 2 range 0.0 .. 4.0;
type Mass is digits 6 range 0.0.. 1.0E35;
```



- ☛ model numbers not equally spaced
- ☛ let compiler worry about implementation

Predefined Floats

- ✓ Float
- ✓ Short_Float
- ✓ Long_Float
- ✓ Short_Short_Float *
- ✓ Long_Long_Float *

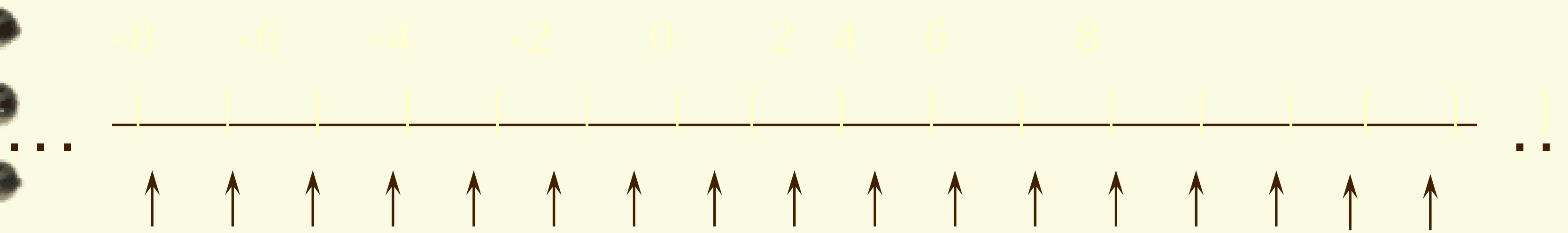
* Optional

Fixed Point Types

- an approximation of real numbers based on the user defined error bound (delta).
- accuracy is the precision of the number to the defined delta.
- Two types
  - Fixed Point (ordinary)
  - Decimal Fixed Point

Fixed Point

- Fixed point uses a fixed distance between consecutive values:



Fixed Point
(ordinary)

- ✓ Use for precise number calculations
- ✓ Multiplication results need to be converted to the type you desire
- ✓ Fixed point numbers can be VERY expensive in terms of conversion. Use with great care, especially in real-time applications.
- ✓ Delta can be any number

Fixed Point Type

(ordinary)

Set of Values:	set of approximations of real Numbers
Structure:	delta N range L.X..U.X
Set of Operations:	assignment :=
	membership in not in
	relation = /= < <= > >=
	math + - * /
	unary + - abs
	exponentiating **

Attributes for Fixed

Small	Delta	Fore	Aft	Address
Base	First	Image	Last	
Machine_Overflows		Machine_Radix		
Machine_Rounds		Max	Min	Pred
Succ	Range	Size	Small	
Valid	Wide_Image		Wide_Value	
Value	Wide_Width		Width	

Decimal Attributes

```
T ' DELTA                -- specified delta value
T ' MANTISSA
T ' SMALL                              -- smallest positive model #
T ' LARGE =  (2.0**T ' MANTISSA-1) * T ' SMALL
                                                   -- largest model #
T ' FORE
T ' AFT
T ' FIRST
T ' LAST
```

Decimal Types

- ✓ Use for precise number calculations
- ✓ Multiplication results need to be converted to the type you desire
- ✓ Fixed point numbers can be VERY expensive in terms of conversion. Use with great care, especially in real-time applications.
- ✓ Delta must be a multiply of ten

Decimal Type

Set of Values: set of approximations of real Numbers

Structure: delta N digits X [range L..U]

Set of Operations:

- assignment :=
- membership in not in
- relation = /= < <= > >=
- math + - * /
- unary + - abs
- exponentiating **

Attributes for Decimals

Small Delta	Fore	Aft	
Address	Base	First	Image
Digits	Round	Scale	
Last	Machine_Overflows		
Machine_Radix	Machine_Rounds		
Max	Min	Pred	Succ
Range	Size	Small	Valid
Wide_Image		Wide_Value	
Value	Wide_Width		Width

Predefined Fixed Point and Decimal Types

- ✓ Duration
- ✓ Money (Annex F)

Currency Example

```
Procedure Main_Driver is
   type Currency is delta 0.01 digits  9 range
                                         0.0..1_000_000.0;
   My_Dollars,
   Your_Dollars : Currency := 0.0;
begin
   My_Dollars     := 100.53;
   Your_Dollars := My_Dollars;
   My_Dollars     := Your_Dollars * 5.0;
   Your_Dollars := -5.03;                    -- error
end Main_Driver;
```

Summary Real Types

- ✓ A representation of actual values (to some precision)
- ✓ a Primitive set of operations
- ✓ a set of attributes to enhance understandability

Summary of Scalar Types

- Two main class
  - Discrete
    - an ordered set of distinct values
  - Real
    - A representation of actual values (to some precision)
- a Primitive set of operations
- a set of attributes to enhance understandability

Summary of Scalar Operators

Operators	Identifiers	Integer	Real	Enumeration
Addition	+ -	X	X	
Explicit Conversion		X	X	X
Exponentiation	**	X	X	
Membership	In not in	X	X	X
Multiplication	*	X	X	
Division	/	X	X	
Qualification		X	X	X
Relational	= /= < ,= > >=	X	X	X
Unary	+ - abs	X	X	
Mod	mod	X		
Rem	Rem	X		

Summary of Types

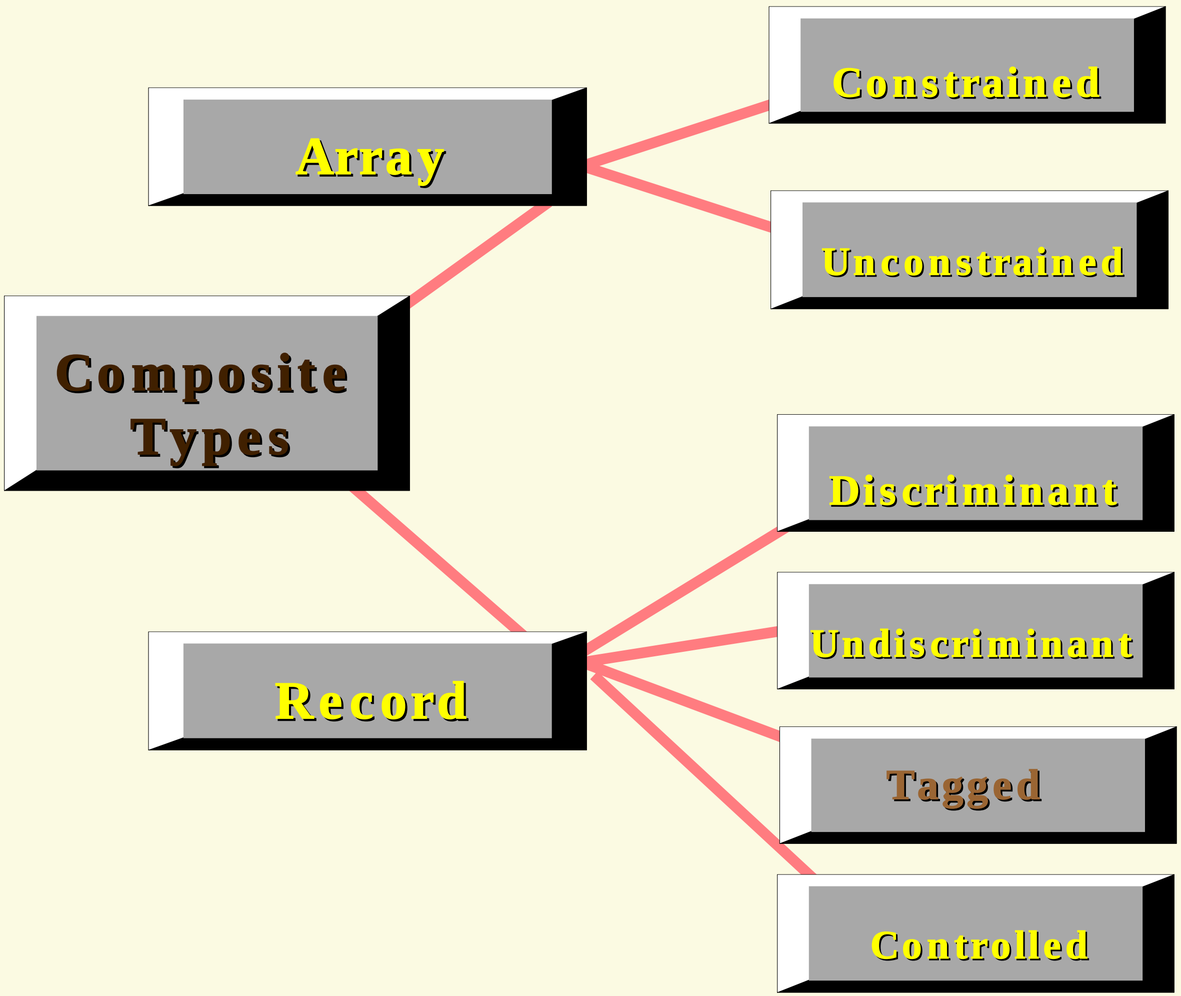
- ✓ Types are use to define the world set you are programming to
- ✓ Typing helps to assist in error prevention and detection
- ✓ Typing enhances maintainability

Composite Types

- Arrays
- Records
- Attributes

Composite Types

Composite Types



Arrays Types

- ✓ Components have the same subtype (homogeneous)
- ✓ Components are referenced by a discreet (enumeration, boolean, or integer) index
- ✓ Kinds of arrays
  - – Constrained
    - limits defined at type declaration
  - – Unconstrained
    - limits defined at object declaration

STRING Literals

- ✓ DELIMITED BY QUOTATION MARKS
- ✓ ANY NUMBER OF CHARACTERS ALLOWED, INCLUDING NONE
- ✓ EXAMPLES:

```
"This is a message"
"first part of a STRING "&
"that continues on the next line"
""For Score ...""        -- YIELDS SINGLE QUOTES
""                       -- YIELDS A NULL STRING
```

Ada STRINGs

type STRING is a composite type

```
type STRING is array (NATURAL range <>) of character;
      -- predefined in package Ada
      STR_5 :  STRING (1 .. 5);
      STR_6 :  STRING (1 .. 6) := "Framus";
      WARNING :  constant STRING := "DANGER";
      subtype TEN_LONG is STRING (1 .. 10);
FIRST_TEN : TEN_LONG := "HEADER     ";
```

STR_6

F	r	a	m	u	s
1	2	3	4	5	6

WARNING

D	A	N	G	E	R
1	2	3	4	5	6

FIRST_TEN

H	E	A	D	E	R				
1	2	3	4	5	6	7	8	9	10

Concatenation (&)

```
STR_A : STRING (1 .. 4) := "Dear";
STR_B : STRING (1 .. 3) := "Ada";
STR_C : STRING (1 .. 10);
...
        STR_C := STR_A  & "  " &  STR_B  &  ",   " ;
```

STR_A

D	e	a	r
1	2	3	4

STR_B

A	d	a
1	2	3

STR_C

D	e	a	r		A	d	a	,	
1	2	3	4	5	6	7	8	9	10

STRINGs Example

```
procedure MAIN_DRIVER is
   NAME            : STRING (1..7);
   Other_Name      : String       := “Cookie”;
begin
  NAME := "JACKSON";
  NAME := "JACK";              -- error
  NAME := "JACKSONS";          -- error
  NAME (1..4) := "JACK";       -- Called a slice
   Name := Other_Name & “ “;
end MAIN_DRIVER;
```

More Array Examples

```
subtype Day_Type is integer range 1..31;

subtype Hours_type is integer range 0..24;

type Month_Type is (Jan, Feb, Mar, Apr, May,
                    Jun, Jul, Aug, Sep, Oct, Nov, Dec);

type Week_Type is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);

type Year_Type is array (Month_Type) of Day_Type;

type Work_week is array (Week_Type range Mon..Fri) of
                                   Hours_type;
```

Multi-Dimensional Arrays

- ✓ The number of dimensions of an array are determined by the number of discrete ranges given in the index constraint.
- ✓ Components of a multi-dimensional array are referenced by an index value for each of the possible indexes in the order they appear in the type definition.

Array Objects

```
Year : Year_Type := (31 , 28, 31, 30, 31, 30, 31,
                      31, 30, 31, 30, 31);
Leap_Year : Year_Type := (Jun => 30, Feb => 29, Sep => 30,
    Apr => 30, Nov => 30, others => 31);
begin
…..
Year (Jul) := 31;
```

Record Types

- ✓ Defines a collection of types that are potentially different (heterogeneous)
- ✓ referenced internals by object name
- ✓ can contain any Ada type or subtype

Record Example

```
-- RECORD TYPE DECLARATION
type DAY_TYPE is range 1..31;
type MONTH_TYPE is (Jan,Feb,Mar,...Dec);
type YEAR_TYPE is range 0..2085;
type DATE_TYPE is
record
        DAY  :DAY_TYPE;
        MONTH:MONTH_TYPE;
        YEAR :YEAR_TYPE;
end record;
-- RECORD OBJECT DECLARATION
TODAY :DATE_TYPE;
```

Record Example (cont)

```
-- RECORD COMPONENT REFERENCE
TODAY.DAY := 15;
TODAY.MONTH := JUN;
TODAY.YEAR := 1990;
if TODAY.DAY = 16 and TODAY.MONTH = June then
        PUT_LINE("Yesterday was 15 June");
end if;

    type WEEK_TYPE is array(1..7) of DATE_TYPE;
    ARRAY_OBJECT : WEEK_TYPE;
    ...
    ARRAY_OBJECT(INDEX).COMPONENT
```

The way to reference a component in
an array of records

Record
More Examples

RECORD OBJECT REFERENCE

```
TODAY := (15,JUN,1990);    -- an aggregate

        -- or --

if TODAY /= (6, DEC, 1942) then ...

        -- or --

TODAY := DATE_TYPE'(15,JUN,1990);

        -- or --

TODAY := (DAY   => 15,
          MONTH => JUN,
          YEAR  => 1990);
```

Default Record Component Values

- ✓ If a component of a record type has a default value, every object declared to be of the record type will have that initial value at declaration time.
- ✓ Can be specified for all or some components
- ✓ At record object elaboration, if no initial value is given, default values are used

Default Record Component Values

```
type DEFAULT_EXAMPLE is
record
    TOTAL :FLOAT :=0.0;
    STATE :STATE_CODE;
    VET   :BOOLEAN :=TRUE;
end record;
SAMPLE:  DEFAULT_EXAMPLE;
```

Records initializing

```
type FRACTION is
  record
    DIVIDEND : DND_TYPE    := 0;
    DIVISOR  : DIV_TYPE    := 1;
end record;

F  : FRACTION;       --  initial value of (0,1)
G : FRACTION  :=  (2,3);
```

Nested Records

- ✓ Components of records may be of any type, including other records
- ✓ The value of a nested record is a nested aggregate
- ✓ Component selection used extended 'dotted' notation

Nested Records Example

```
type TEMPERATURE_LOG is
record
      TEMP:INTEGER;
      DATE:DATE_TYPE;
end record;
LOG:TEMPERATURE_LOG;
...
LOG.TEMP :=50;
LOG.DATE.DAY :=15;
LOG.DATE.MONTH := JUN;
LOG.DATE.YEAR := 1990;
```

```
   -- or
LOG.DATE :=(15,JUN,1990);
-- or even
LOG:= (TEMP => 50,
            DATE => (15,JUN,1990));


-- or, using positional notation
LOG := (50, (15,JUN,1990));
```

Records

```
type DAYS is (MON, TUE, WED, THU, FRI, SAT, SUN);
type DAY_TYPE is range 1..31;
type MONTH_TYPE is (Jan,Feb,Mar,...Dec);
type YEAR_TYPE is range 0..2085;

type NEW_DATE_TYPE is
record
        DAY_OF_WEEK : DAYS;
        DAY : DAY_TYPE;
        MONTH : MONTH_TYPE;
        YEAR : YEAR_TYPE;
end record;
TODAY : NEW_DATE_TYPE;

begin
        TODAY.DAY_OF_WEEK := FRI;
        TODAY.DAY := 15;
        TODAY.MONTH   := JUN;
        TODAY.YEAR := 1990;
```

Records

- ✓ No discriminant
  - – Components do not depend on a discriminant
- ✓ Discriminant
  - – Components of a record depend on another component called a discriminant

Record Variant Parts

- ✓ The actual existence of certain fields can depend on a discriminant value

```
type DRIVER is (GOOD,BAD);
type INSURANCE_RATE is range 1..5000;
type DISCOUNT is delta 0.001 range 0.0..1.0;
type INSURANCE (KIND:DRIVER) is
record
        NORMAL_RATE:INSURANCE_RATE;
        case KIND is
                when GOOD => DISCOUNT_RATE:DISCOUNT;
                when BAD => ADDITIONAL:INSURANCE_RATE;
        end case;
end record;
```

More Examples

```
A_DRIVER : INSURANCE (GOOD);
ANOTHER : INSURANCE (BAD);

begin
    A_DRIVER.NORMAL_RATE        := 250;
    A_DRIVER.DISCOUNT_RATE := 0.015;

    ANOTHER.NORMAL_RATE         := 250;
    ANOTHER.ADDITIONAL          := 1000;
```

Other Record Types

- Tagged
  - Used when you need to expand the data structure in different ways
- Controlled
  - Used when you want to do automatic set-up and clean-up of data structures

Record Summary

- ✓ NO DISCRIMINANT
- ✓ DISCRIMINANT
  - components of the record depend on a component called a discriminant
- ✓ VARIANT
  - record structures of the same type contain different components, based on discriminant

Record with a Variant part

```
with Ada.Text_Io, Ada.Strings.Bounded, Ada.Calendar;
 procedure Variantr is
   Max_Buffer :  constant Positive := 256;
package Message_Buffer is new
ada.Strings.Bounded.Generic_Bounded_Length (Max_Buffer);
   type Message_Classifications is (Unclass,Flash, Alpi);
      type Messages (Mesg_Class : Message_Classifications := Unclass) is
   record
      Buffer : Message_Buffer.Bounded_String;
      case Mesg_Class is
         when Unclass =>
            Save : Boolean := True;
         when Flash =>
            Time_Stamp : Ada.Calendar.Time;
                 when Alpi =>
            Flag_Officer : Boolean := True;
end case;
   end record;
    Buf : Message_Buffer.Bounded_String;
   TS : Ada.Calendar.Time;

M1 : Messages;
M2 : Messages ( Mesg_Class => Flash);
```

```
begin
   M1 := (Mesg_Class=>Alpi, Buffer => Buf, Flag_Officer => False);
   M1 := (Mesg_Class=>Unclass, Buffer => Buf, Save => False);
   M1.Save :=  False;

M2 := (Mesg_Class=>Unclass, Buffer => Buf, Save => False); --Discriminant
check will fail at run-time, exception will be raised

M1.Flag_Officer :=  False; --  CONSTRAINT_ERROR

null;
end VariantR;
```

Composite Types

- ✓ Arrays
  - Homogeneous collection
  - indexed by a scalar object
- ✓ Records
  - heterogeneous collection
  - referenced internals by object name

Exercises

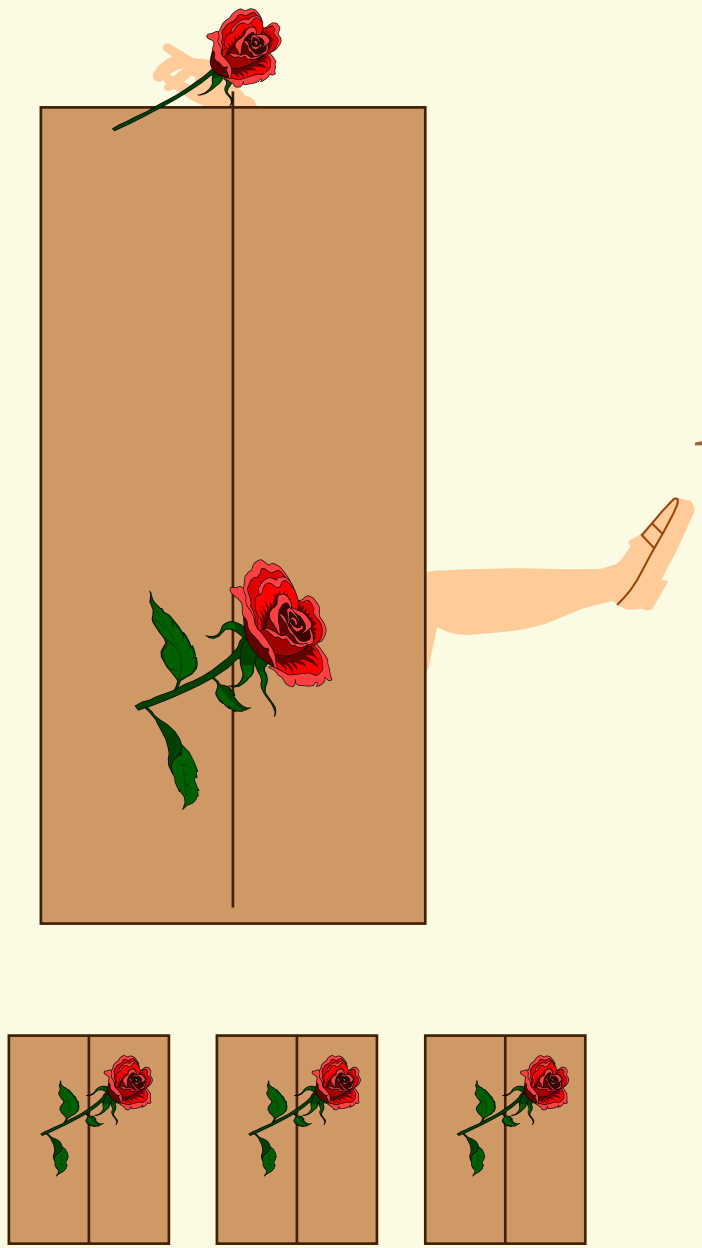
Objective: Demonstrate the use of records, arrays, and loops in an Ada program.

Problem Write a program to count the number of vowels (A,E,I,O or U) in its input. Allow the user to type or read from a file, sequences of characters (as many as they like) terminate by a null string or end of file. Display the number of occurrences of each vowel as well as a grand total.

Source Code

```
 --Filename ex1b.adb
with Ada.Text_Io;
--with Enumeration_Io;
procedure ex1b is
 type Counters is range 0..1_000_000;

type Vowels is (‘A’, ‘E’, ‘I’, ‘O’, ‘U’);

type Vowels_Count is
record
		Vowel : String := ‘A’;
		Count  : Counters := 0;
End record;

type Tables is Array (Vowels’First..Vowels’Last) of Vowels_Count;   -- Change the area index to ‘range

Table : Tables :=(
begin   --Main
		Ada.Text_Io.Put (" ");



end ex1b;
```

Ada’s Private Parts



What is essential is invisible to the eye.

Antoine de Saint-Exupery, The Little Prince

Ada’s Private Parts

Information hiding -- Avoid revealing implementation details. The more that is hidden the more freedom you will enjoy in maintenance.

Private -- *outside view, predefine = /=*

Limited Private -- *outside view, only user define, no :=*

Ada's Private Parts

Example

You need to change the way dates are represented to make year 2000 and beyond calculation easier. Change dates storage from day, month, year to Julian date.

```
package SAC.Date_Util is
        subtype DAYS is INTEGER range 1..31;
        type  MONTHS is (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
        subtype YEARS is Integer range 1901 .. 3000;
        type WEEKDAYS is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
        type DATES is
        record
                Day     : DAYS;
                Month : MONTHS;
                Year    : YEARS;
        end record;
function Weekday (Date : DATES) return WEEKDAYS;
function Valid (Date : DATES) return Boolean;
end SAC.Date_Util;
```

Ada's Private Parts Example

```
package SAC.Date_Util is
    subtype DAYS is INTEGER range 1..31;
    type  MONTHS is (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
    subtype YEARS is Integer range 1901 .. 3000;
    type WEEKDAYS is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
    type DATES is private;
    function Day (Date : DATES) return DAYS;
    function Month (Date : DATES) return MONTHS;
    function Year (Date : DATES) return YEARS;
    function Weekday (Date : DATES) return WEEKDAYS;
    function Date (Day : DAYS; Month : MONTHS; Year : YEARS) return DATES; --
    Constructor
    Date_Error  :  exception;
    private    -- visible part ends here
    type DATES is
    record
        Day     : DAYS;
        Month : MONTHS;
        Year    : YEARS;
    end record;
end SAC.Date_Util;
```

Private is Better!

Ada's Private Parts

Example

```
generic
   type items is private;
package test is
      type STACKS is limited private;

--    procedure Push (Stack : in out STACKS; Item : in ITEMS);
--
--    procedure Pop (Stack : in out STACKS; Item : out ITEMS);
--
--    function Empty (Stack : STACKS) return BOOLEAN;
--
--    function Full (Stack : STACKS) return BOOLEAN;
--
   Stack_Overflow , Stack_Underflow  :  exception;

private
    type STACK_ITEMS;
   type STACKS is access Stack_ITEMS;
end test;
```

```
package body Test is
         Max_Items : constant := 10;

type STACK_ARRAY is array (1..Max_Items) of ITEMS;

subtype Stack_Pointer is NATURAL range 0..Max_Items;

type STACK_ITEMS is   -- Completion of type declaration
         record
            Value : STACK_ARRAY;
            Top   : STACK_POINTER :=0;
         end record;
    end Test;
```

More Private is Better!

Opaque type --name is visible but definition is hidden inside the package body

Ada's Private Parts

Private child Packages

```
package Sorite_Strike is

   type DATAS is abstract tagged private;
   type TARGETS is private;
   type PRECENTS is digits 5 range 0.00 .. 100.00;

   procedure Probability_Damage (Target : in TARGETS;
      Data : in DATAS;  Precent : out PRECENTS) is abstract;

   function  Valid (Target : in TARGETS; Data : in DATAS)
                                  return BOOLEAN is abstract;

   private
    type DATAS is abstract tagged
      record
         Vehicle : STRING (1..15);
      end record;

     type TARGETS is range 100..999;

end Sorite_Strike;
```

```
private package Sorite_Strike.Data_Source is

  type DATA_Record is private;

 procedure Probability_Damage (Target : in TARGETS;

    Data : in DATA_Record; Precent : out PRECENTS);

 function  Valid (Target : in TARGETS; Data : in
                    DATA_Record) return BOOLEAN;

  private

   type DATA_Record is  new Datas with

   record

     Unit_Number : Integer ;

     Unit_ID     : String (1..15);

  end record;

end Sorite_Strike.Data_Source;
```

More Private is Better!

Exercise

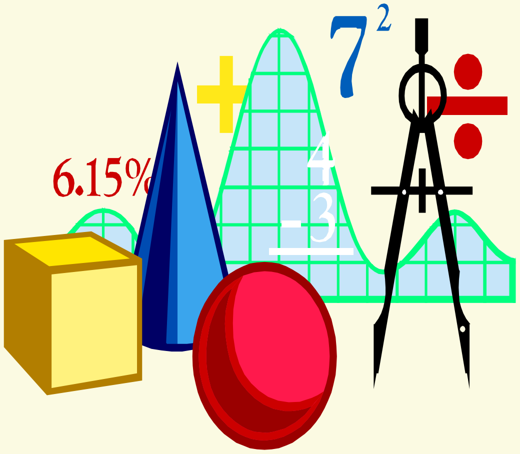
- Modify your previous program.
- Change Age to Private.   Pass Age between subprograms.
- Modify Age objects.
- Try adding and multiplying Ages;
- Change Age to Limited Private and *, / operations.

Ada-Specific Design Considerations

- Misuses of the Language
- Designing for Portability.
- Designing for Reusability

Object-Oriented Programming

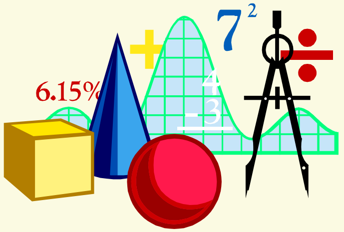
- Hierarchical Library Units
- Tagged types
- Class Wide programming
- Dispatching
- Control



What a language needs for OOP

Essentials	Nice to have
Abstraction	Typing
Encapsulation	Concurrency
Modularity	Persistence
Hierarchy	

Ada 95 and OOSE

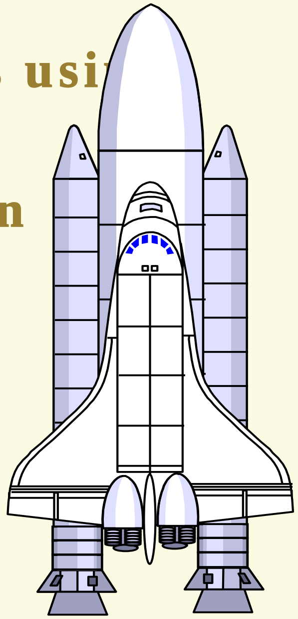


Ada 95

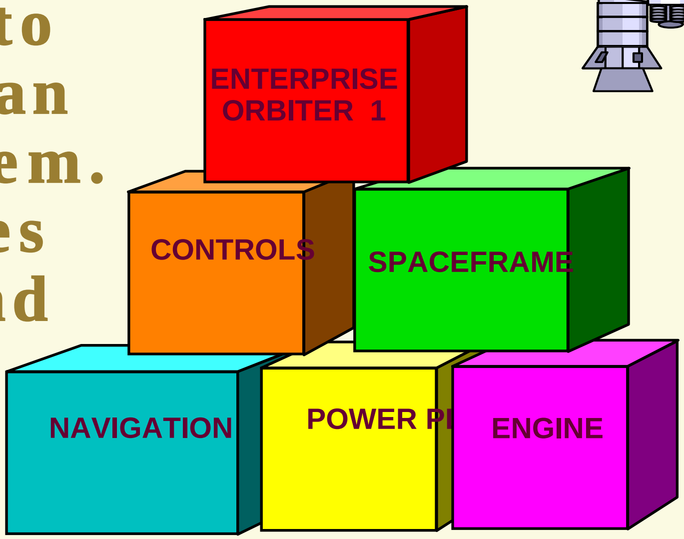
- Is a superior language for supporting software engineering principles.
- Is a language which provides support for OOP for those who wish to use it.
- Provides a somewhat different model for OOP than other languages.

Object-Oriented Methodologies

- OO methodologies attempt to build complex systems usi[ng] "classes" and "objects" as the building blocks
  — Attempts to model the real world via abstraction
  — Evolutionary, not revolutionary



- Experience has shown that this is the superior way to manage and decompose an inherently complex system. Functional methodologies require you to understand the entire system before you can modularize it!



w to implement the OO Paradigm

Macro View

- Identify the core requirements for the software (conceptualization)
- Develop a model of the system's desired behavior (analysis)
- Create an architecture for the implementation (design)
- Evolve the implementation through successive refinements (evolution)
- Manage postdelivery evolution (maintenance)

How to implement the OO Paradigm

Micro View

- Identify the classes at a given level of abstraction (OOD issue)
- Identify the objects at a given level of abstraction (OOD issue)
- Identify the relationships among these classes and objects (OOD issue)
- Specify the interface and then code the implementation of these classes and objects (OOP issue).

Benefits of the OO Paradigm

Reuse

- Well-designed classes can be extended via inheritance. Previous code can be reused without modification to users of the original code.
  - Allowing lots of subclasses allows special cases to be truly special without modifications to normal classes.

Integration

- OO supports incremental development.
  - Additions to the system are easier.
  - It is easy to develop the parent class first, and then concentrate on subordinate classes.

Benefits of the OO Paradigm (cont.)

Testing

- Easier to test, as specific behaviors and classes can be tested individually.
  - Specific test cases can be designed to test a class, and then this behavior does not need to be retested for subordinate classes

Maintenance

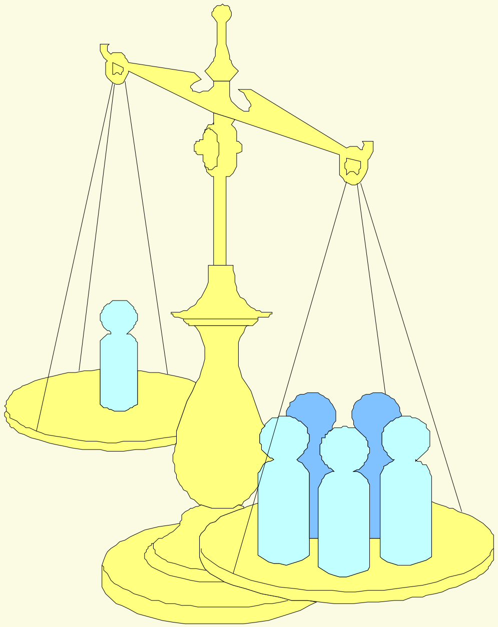
- Additions to the system can be accomplished without modifying existing code.
  - Instead of modifications to existing classes, new classes can be constructed

Ada 95

- Ada 95 core language =

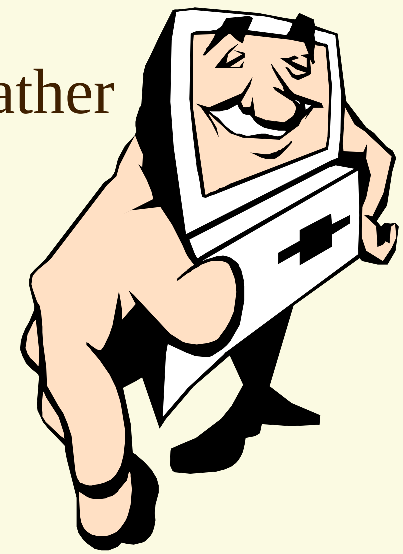
  Ada 83

  + OOP(inheritance, Polymorphism, Dynamic Binding)
  + Hierarchical library units
  + data-oriented synchronization
  + flexible scheduling
  + … Specialize needs annexes



Ada 95 and OOSE

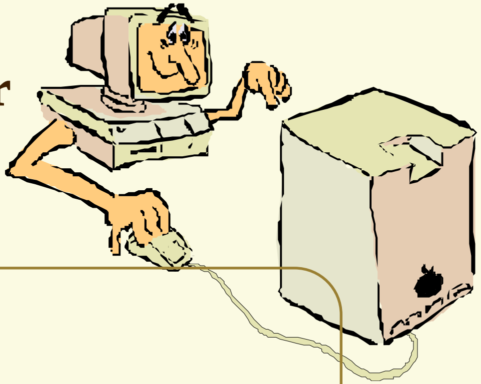
- ✓ Ada 95 has attempted to build on Ada 83 rather than changing the syntax drastically
- ✓ Mechanism for inheritance has been added
  - Permits type extension
  - Uses tagged types



- Child library units provide different views to different clients
- Ada 95 provides full support for OOP (Object-Oriented Programming)

Extensible vs. non-extensible types

Sensor



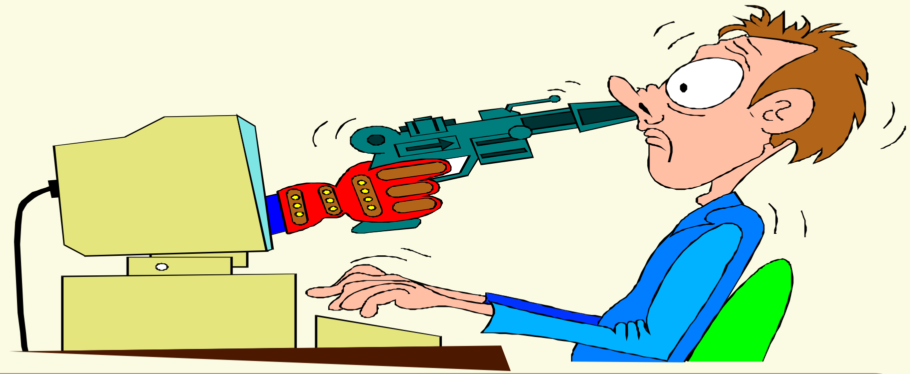
```
type Sensor is  record -- Not Extensible
            Current_State : State := On;
end Record;

procedure XXX ( Some_Parameter : Sensor);
function YYY return Sensor;
function ZZZ (Some_Parameter : in Sensor) return WWW;
```

This is an Abstract Data Type (ADT).

You can extend the functionality by adding operations. The data structure is static.

Extensible vs. non-extensible types

Sensor?



```
type Sensor is  tagged record --  Extensible
            Current_State : State := On;
end Record;

procedure XXX ( Some_Parameter : Sensor);
function YYY return Sensor;
function ZZZ (Some_Parameter : in Sensor) return WWW;
```

This is a tagged type that can be used as the parent of an inheritance chain.

The data structure may be extended, and operations added.

Encapsulation

- Declare a class
- Attributes
- Methods (Operations)
- An object's state based on attributes
- An object's behavior based on methods
- Packaged public & private components

✓ Encapsulation

☑ Inheritance

☑ Polymorphism

Encapsulation...

- ...is sometimes used as a synonym for “information hiding” [Meyer97]
- ...encompasses a classes attributes and operations [Pressman01]
- ...the combination of hiding and message inter-face abstraction; of hiding and packaging [Eliens00]
- ...extends the concept of abstraction from data to both data and functions [Daconta99]
- ...the basic idea behind a package or a module [Ledgard96]

Encapsulation

- Declare a class
  - Attributes
  - Methods (Operations)
  - An object's state based on attributes
  - An object's behavior based on methods
  - Packaged public & private components

✓ Encapsulation
☑ Inheritance
☑ Polymorphism

Declaring a Class: Ada 95

```
package Rx is
type Rx_type is private;
procedure del_Rx (Rx_num);
procedure refill_Rx (Rx_num);
procedure wr_Rx (Rx_num, patient,
physician,
                 ...);
private:
  type Rx_type is record
    Rx_num: int;
    patient: string;
    physician: string;
    med_name: string;
    num_refills: integer;
  end record;
end Rx;
```

Private Parts: Pkg Body

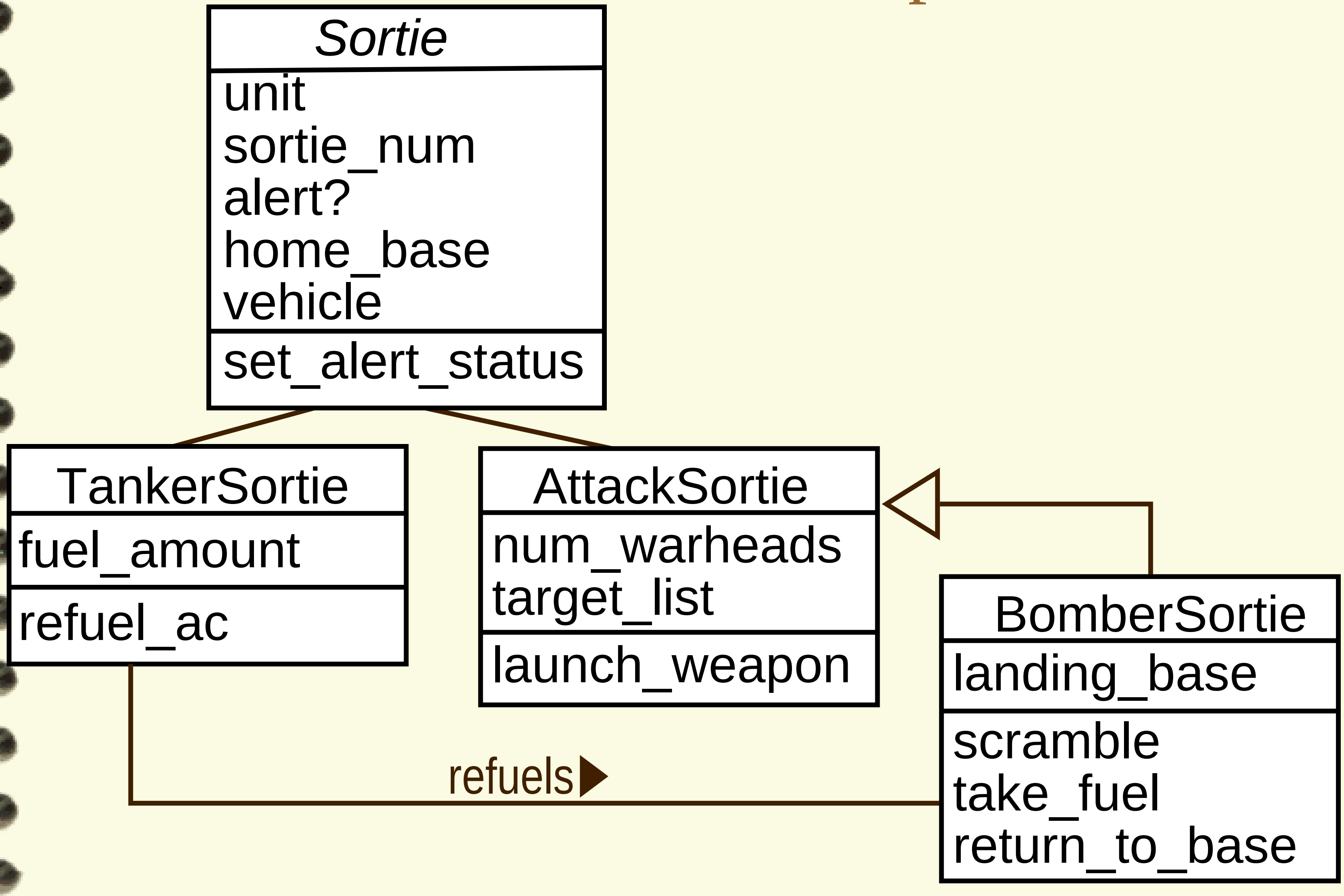
```
package body Rx is

procedure refill_Rx (the_num : in int) is
Rx : Rx_type;
begin
  --traverse Rx store, find Rx record
  if Rx.Rx_num = the_num then
    if Rx.num_refills > 0 then
      Rx.num_refills := Rx.num_refills - 1;
    else
      raise no_refills_left;
    endif;
  endif;
end refill_Rx;
```

Inheritance

- Declare a parent class
  - May be abstract
  - Can have children
  - Children inherit ALL attributes & ops
  - Children may also have children

☑ Encapsulation
✓ Inheritance
☑ Polymorphism

Inheritance Example



Inheritance: Ada 95

```
type attack_sortie is new sortie with
  record
    num_warheads: integer;
    target_list: target_list_type;
  end record;

type bomber_sortie is new attack_sortie with
  record
    landing_base: string;
  end record;
```

Inheritance: Ada 95

```
type attack_sortie is new sortie with
  record
    num_warheads: integer;
    target_list: target_list_type;
  end record;

type bomber_sortie is new attack_sortie with
  null record;
```

Tagged Types

```
type Rectangle is tagged
    record
        Length : Float := 0.0;
        Width   : Float := 0.0;
     end record;
--  Operations for inheritance now defined
--   Example: Rectangles have a defined perimeter, and
--   children derived from Rectangle will have Perimeter

function Perimeter (R : in Rectangle ) return Float is
  begin
     return 2.0 * (R.Length +R.Width);
  end Perimeter;
```

Tagged Types - Inheritance

```
type Square is new Rectangle with
    null record; -- inherit from parent, add no new fields

--Square will inherit the primitive operation Perimeter

function Area (S : in Square ) return Float is
  begin
    return (S.Length  * S.Width);
  end Area;
```

Any operation declared in the scope of a tagged type (or in the scope of a type derived from a tagged type) is a “primitive” operation. It is automatically inherited.

This is an example of “extending” the type without adding any new components. This is similar to the way Ada 83 used derived types.

Tagged Types - Inheritance


```
type Cuboid is new Rectangle with
    record
      Height : Float := 0.0;
     end record;

function Perimeter (C : in Cuboid ) return Float is
begin
   return Perimeter (Rectangle(C)) * 2 + ( 4 * C.Height);
end Perimeter;
```

Cuboid inherits Perimeter from Rectangle. The function will have to be updated for the new type (Perimeter is defined differently for cubes!).

To do this, you need to *override* the operation. One way to do this is to write a new Perimeter. A better way it to base the new Perimeter on the parent class operation. Note the *view conversion* in Perimeter - preventing recursion, and basing the perimeter of a cube on the perimeter of the base class.

Ada 95 Inheritance Programming

```
with Calendar;

package New_Alert_System is

    type Device is (Teletype, Console, Big_Screen);

        --define the "base class"

    type Alert is tagged
        record
            Time_of_Arrival : Calendar.Time;
            Message : Text;
        end record;
    procedure Display ( ...
    procedure Handle (A : in out Alert);
    procedure Set_Alarm ( ...
```

Ada 95 Inheritance Programming

```
--now, use the base class and extend it via inheritance

type Low_Alert is new Alert;

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

--and override one of the inherited operations

procedure Handle (MA : in out Medium_Alert);

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

procedure Handle (HA: in out High_Alert);

end New_Alert_System;
```

Completing the subprograms...

```
package body New_Alert_System is
    procedure Handle(A: in out Alert) is begin
            A.Time_of_Arrival:= Calendar.Clock;
            Log(A);
            Display(A, Teletype);
    end Handle;
    procedure Handle(MA: in out Medium_Alert) is begin
            Handle(Alert(MA) );  -- conversion(no dispatch)
            MA.Action_Officer:= Assign_Volunteer;
            Display(MA, Console);
    end Handle;
procedure Handle(HA: in out High_Alert) is begin
            Handle(Medium_Alert(HA) );  -- conversion(no dispatch)
            Display(HA, Big_Screen);
            Set_Alarm(HA);
    end Handle;
...
end New_Alert_System;
```

Another user needs to extend again ...

```
with New_Alert_System;
package Emergency_Alert_system is
    type Emergency_Alert is
         new New_Alert_System.Alert with private;
   procedure Handle (EA : in out Emergency_Alert);
   procedure Display (EA : in Emergency_Alert;
                                  On : in New_Alert_System.Devices);

procedure Log (EA : in Emergency_Alert);

private
   ....
end Emergency_Alert_System;
```

Abstract Types & Subprograms

```
-- Baseline package used to serve as root of inheritance tree
package Vehicle_Package is
   type Vehicle is abstract tagged null record;
   procedure Start (Item : in out Vehicle) is abstract;
end Vehicle_Package;
```

- Purpose of an abstract type is to provide a common foundation upon which useful types can be built by derivation.
- An abstract subprogram is a place holder for an operation to be provided (it may not have a body).
- An abstract subprogram MUST be overridden for EACH subclass before any object of the subclass can be declared.

Abstract Types and Subprograms

```
type Train is new Vehicle with
    record
         passengers : Integer;
    end Train;

My_Train : Train;                    --  ILLEGAL
```

We can’t yet declare an object of *Train*. Why? Because we haven't filled in the *abstract parts* declared in its parent. We have completed the *abstract record*, but still need to define procedure *Start* for the *Train*.

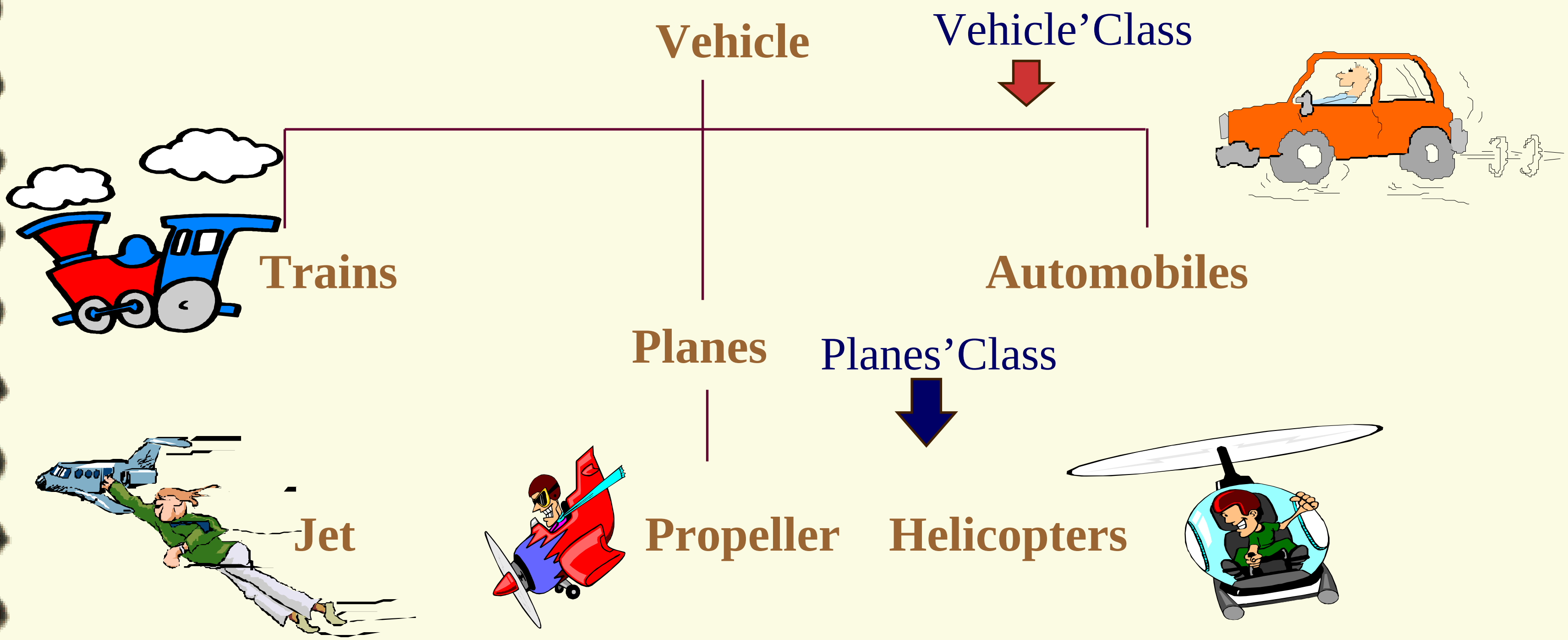
```
type Train is new Vehicle with
    record
         passengers : Integer;
    end Train;

procedure Start (Item : in out Train) is ....
My_Train : Train;
```

Abstract types

```
type Planes is abstract new Vehicle with
        record
                Wingspan : Some_Type;
        end Planes;

function Runway_Needed_To_Land
                    (Item : Planes) return Feet is abstract;
```

Type Planes is an abstract type based on Vehicle, which is also an abstract type. Therefore, the procedure Start must be overridden (to satisfy the requirements for Vehicle) and the function Runway_Needed_To_Land must be overridden (to satisfy the requirements for Planes) for any type derived from Planes.

Class Wide Programming type T'Class



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

Polymorphic Data

- Class-wide variables are said to be polymorphic because their values are of different “shapes” (from the Greek poly, many, and morphe, form).
- When calling a subroutine with a polymorphic parameter, the compiler is unable to determine which subroutine to call, since the actual type of a polymorphic variable will not be known until run time!
- Thus, the binding to some subroutines must be delayed until execution - hence the term “dynamic binding”.
- A call to a routine that must be dynamically bound is called a “dispatching” call

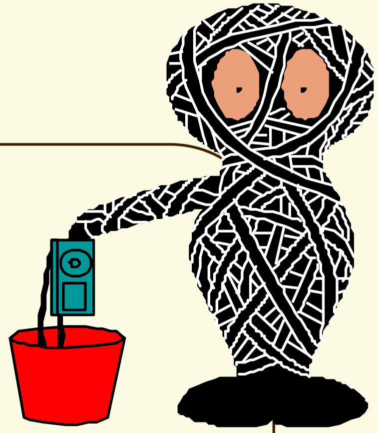
Class Wide Programming (Dispatching)

```
-- class-wide value as parameter
Procedure Move_All ( Item : in out Vehicle'Class) is
...
begin
   ...
   Start (Item);          -- dispatch according to tag
   ...
end Move_All;
```

The procedure Move_All is a ***class-wide*** operation, since any variable in the Vehicle hierarchy can be passed to it.

Start, however, is defined for each type within the Vehicle hierarchy. Depending on the type of Item, a different Start will be called. During runtime, the specific type of Item is known, but it is not known at compile time. The ***runtime system*** must ***dispatch*** to the correct procedure call.

Static Binding



```
-- class-wide value as parameter

Procedure Move_All ( Item : in out Vehicle'Class) is
...
begin
    ...
    Start (Item);          -- dispatch according to tag (Dynamic Dispatching)
                 --vs.
    if Item in Jet then
       Start (Jet(Item));    -- static call to the Start for Jet.   .
                                      -- This is a membership test and View Conversion.

    ...
end Move_All;
```

Problems with Dynamic Binding

- ✓ It lengthens execution time
  - Each call causes slight time delay
    - Tag of the variable must be examined
    - Correct subprogram must be found by searching inheritance tree from the "tag" upwards
- ✓ It is non-deterministic
  - I can bound it, but only as an upper limit or average case
- ✓ It makes verification difficult, as you can't formally define which subroutine you are actually calling

Dynamic Binding Ada 95 does it better!!

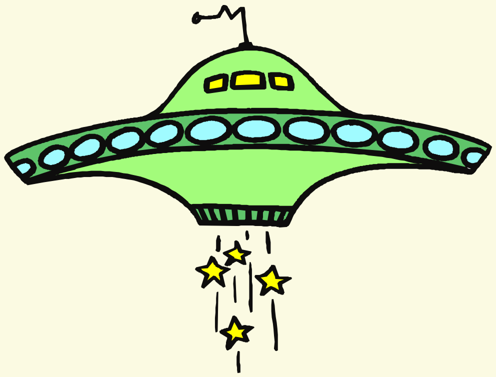
- Note that under Ada 95 and it's library, all dispatching calls are guaranteed to find a subprogram at run time
- Dispatching calls are only made when a procedure that has accepted a class-wide parameter passes this same parameter off to a non class-wide procedure

Actual Parameter Type	Formal Parameter Type: Specific	Formal Parameter Type: Class-Wide
Specific	static binding	class-wide procedure
Class-Wide	dispatching call	class-wide procedure

Class Wide Programming (Dispatching using pointers)

```
-- Vehicles held as a heterogeneous list using an access type.
GIVEN:  type Vehicle_Ptr is access all Vehicle'Class;
```

```
--control routine can manipulate the vehicles directly from the list.
procedure Move_All is
    Next : Vehicle_Ptr;
begin
    ...
    Next :=  Some_Vehicle;   --  Get next vehicle
    ...
    Start (Next.all);              --  Dispatch to appropriate Handle
    ...                            --  Note the dereferencing of pointer
end Move_All;
```



Dynamic Selection- Referencing a subprogram

An access type can refer to a subprogram; an access-to-subprogram value can be created by the 'Access attribute and a subprogram can be called indirectly by dereferencing such an access value.

```
type Trig_Function is access function (F : Float) return Float;
T : Trig_Function;
X, Theta : Float;
T := Sin'Access;
X := T(Theta);
X := T.all (Theta);
```

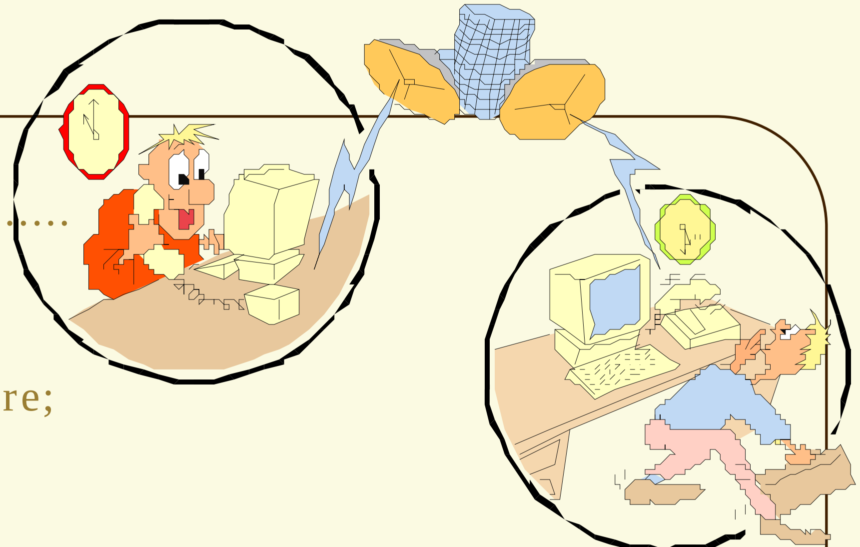
T can "point to" functions such as Sin, Cos and Tan

Indirectly call the subprogram currently referred to

Calling a subprogram via a pointer (and default parameters)

```
type Message_Procedure is access procedure(M : in String := “Error!”);

procedure Default_Message_Procedure (M : in String);

Give_Message : Message_Procedure := Default_Message_Procedure’Access;
...
procedure Other_Procedure (M : in String);
...
Give_Message := Other_Procedure’Access;
...
Give_Message (“File not found.”);      --calls what Give_Message is pointing to
Give_Message.all;                      --ditto, but uses default!
```

Callbacks



```
procedure Call_Word_Processor is......

procedure Ring_Bell is ..........

type Action_Call is access procedure;

type Button_Type is
      record
            X_Pos : Integer;
            Y_Pos : Integer;
            Action_When_Left_Button_Pushed    : Action_Call;
            Action_When_Right_Button_Pushed  : Action_Call;
      end;

Button_1 : Button_Type := ( 100,  50,  Call_Word_Processor'access,
                                    Ring_Bell'access );
```

Constructors/Destructors (Controlled types in Ada95)

```
package Ada.Finalization is   --system defined package

type Controlled is abstract tagged private;

procedure Initialize  (Object: in out Controlled);
procedure Adjust      (Object: in out Controlled);
procedure Finalize    (Object: in out Controlled);
```

Initialize- **implicitly called after storage is allocated, and also after any default initialization.**

Adjust- **implicitly called after a *copy* is performed. All tagged types are passed by reference. Thus, Adjust is called for function return and assignment.**

Finalize- **implicitly called after object becomes inaccessible (leaving scope, deallocation of object) or before assignment.**

Multiple Inheritance

- ✓ There is no built-in multiple inheritance mechanism in Ada 95
  - – limited usefulness in language with good module and child libraries
  - – adds overhead to processing
  - – typically used to “program around” rather than to create a true hierarchy from several classes
- ✓ It can be simulated in two ways
  - – sibling inheritance - creating a record with components of each type you want to inherit from
  - – mix-in inheritance - using generics to extend a type

Mixing Inheritance

Create a generic that takes a tagged type as a parameter

```
generic
        type Parent_Type is tagged private;
package  Mixin is
…..
        type Mixed_Type is new Parent_Type with private;

end Mixin_Package;
-----------------------------------------------------------------------------
type Original_Type is tagged private;
package Mixin_Package is new Mixin (Original_Type);
```

Type Mixin_Package.Mixin inherits from both Original_Type and from Mixed_Type. It is a member of both Original_Type'Class and Mixed_Type'Class.

Program Libraries

The Ada program library brings important benefits by extending the strong typing across the boundaries between separately compiled units.

Unfortunately, the flat nature of the Ada 83 library gave problems of visibility control; for example it prevented two library packages from sharing a full view of a private type.

A common consequence of the flat structure was that packages become large and monolithic. This hindered understanding and increased the cost of recompilation.

A more flexible and hierarchical structure was necessary.

Hierarchical Libraries

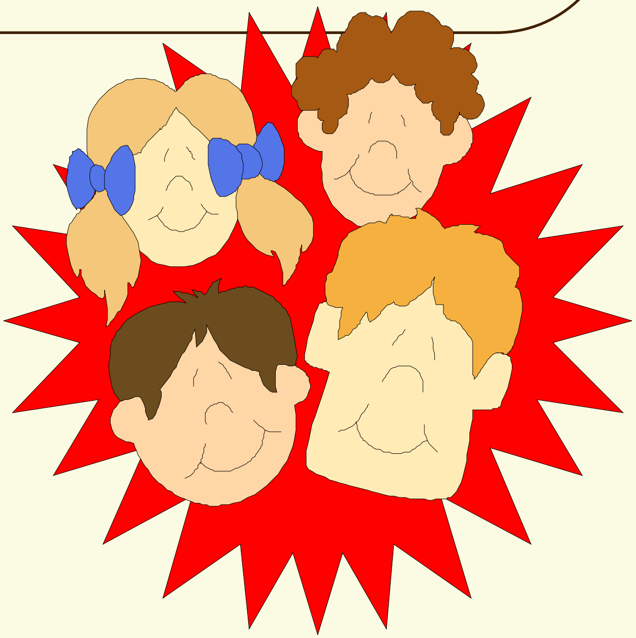
```
package Complex_Numbers is
    type Complex is private;
    function "+" (Left, Right  : Complex) return Complex;
    ... -- similarly "-", "*" and "/"
    function Cartesian_To_Complex (Real,Imag:Float) return Complex;
    function Real_Part (X : Complex) return Float;
    function Imag_Part (X : Complex) return Float;
private
....
end Complex_Numbers;
```

Problem : some users of the above package need additional functionality - they need Polar Coordinates. Adding additional functions to the package requires all users to recompile, and adds functionality to ALL users, even those who don't need it.

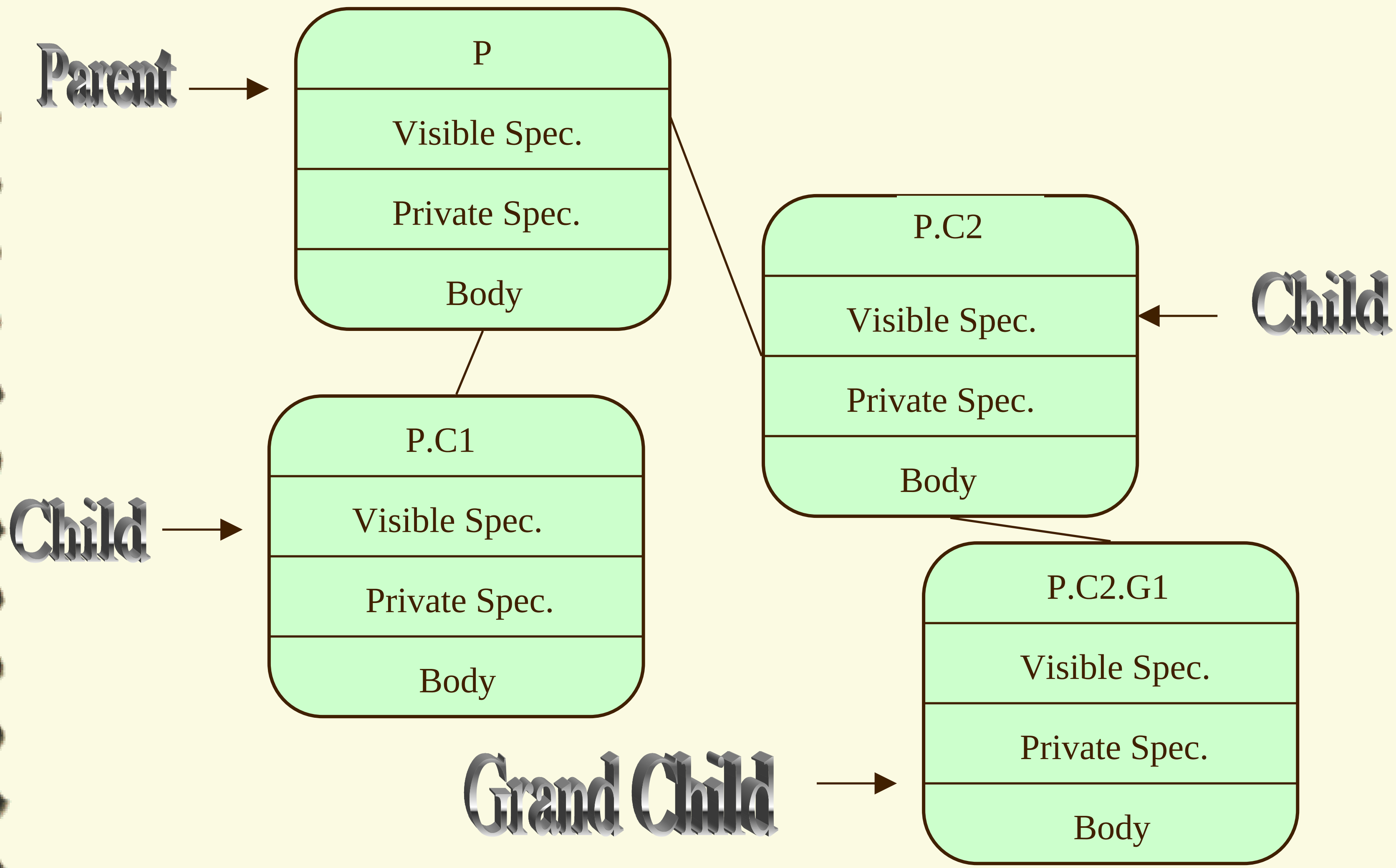
Public Children

```
package Complex_Numbers.Polar is
    procedure Polar_To_Complex (R, Theta : Float) return Complex;
    function “abs” (Right : Complex ) return Float;
    function Arg ( X : Complex) return Float;
end Complex_Numbers.Polar;
```

Solution - create a public child. Only those users who specify “with Complex_Numbers.Polar” gain the functionality. Other users are unaffected (and need not recompile).



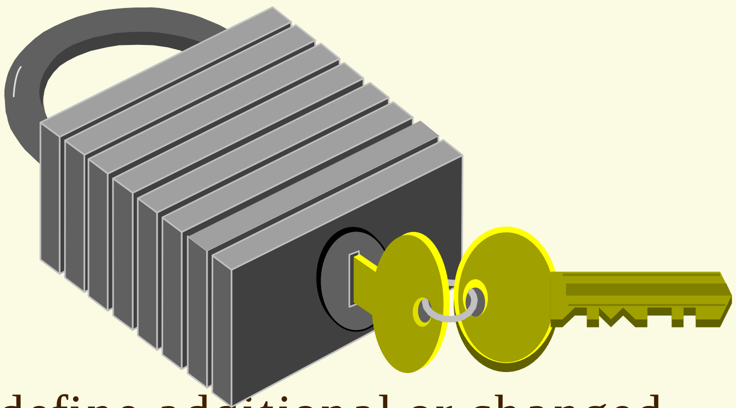
Visibility Rules of a Child Package



Meyer's Modules

✓ Closed Module

- is executable, its behavior is well defined, it's been V&V.

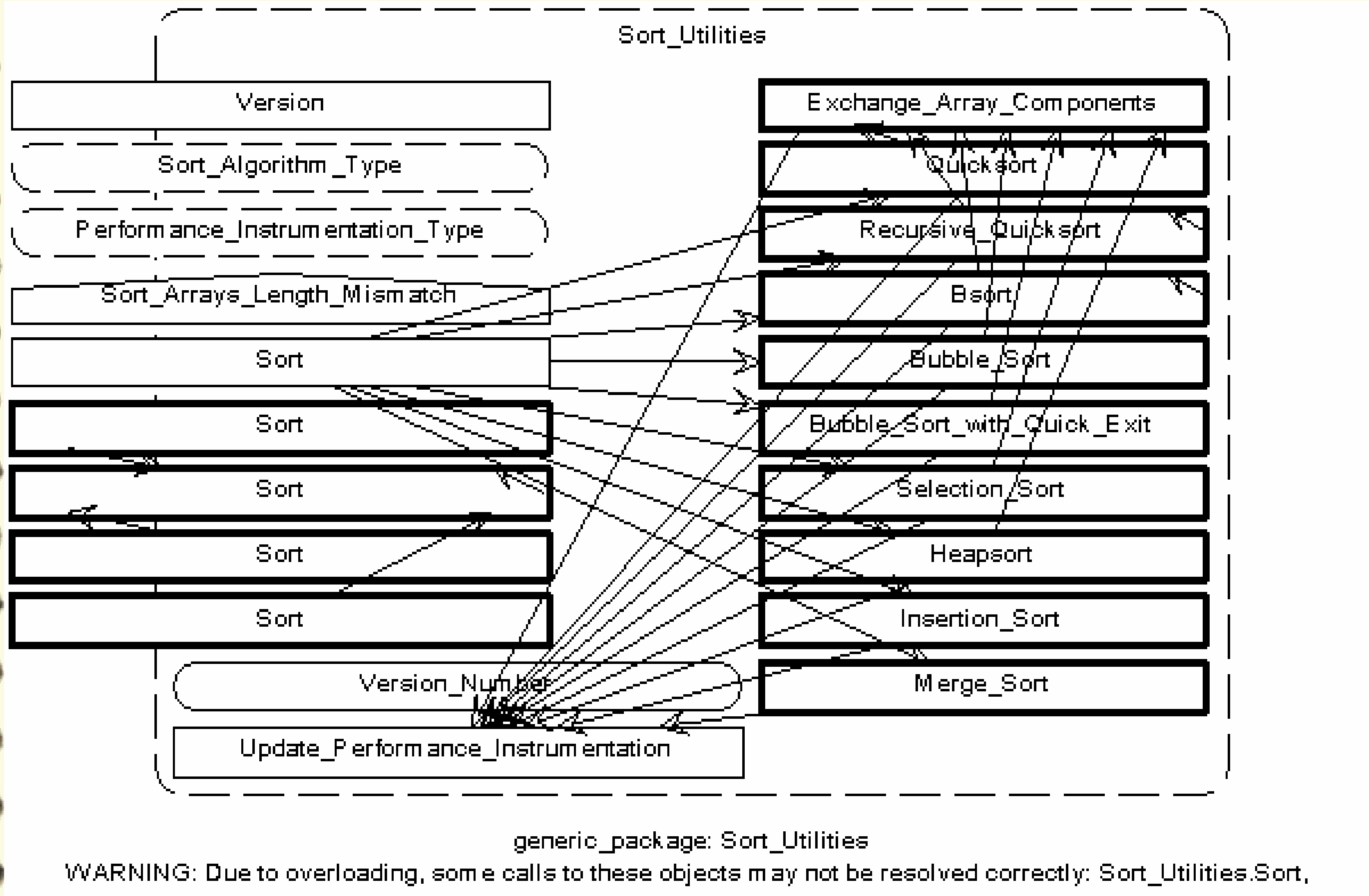


✓ Open Module

- is extendable, it's possible to define additional or changed functionality with out changing the close module.

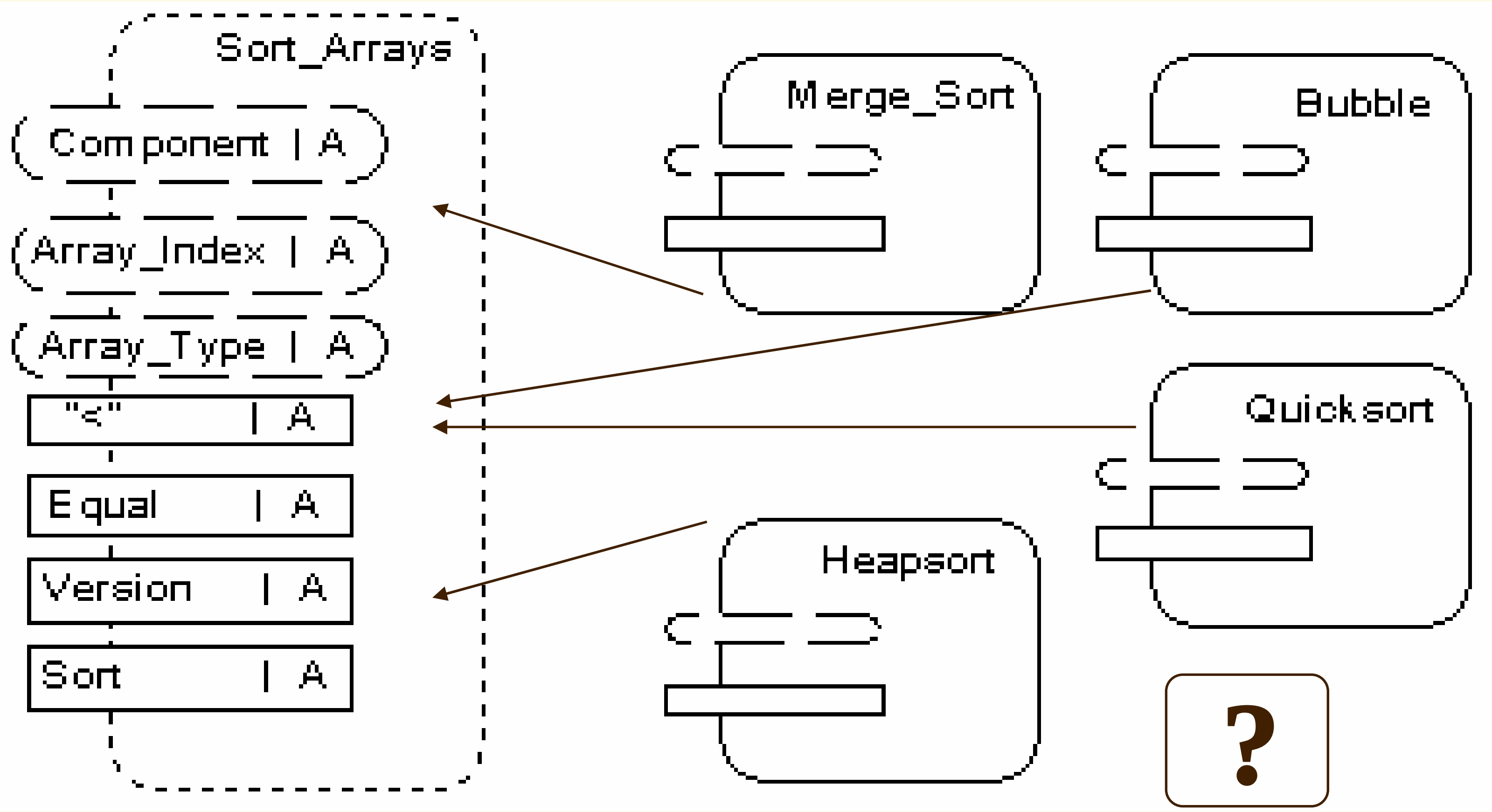
Closed Module + Open Module = Inheritance

Closed Module



Abstract Type

Open Module



Hierarchical Libraries

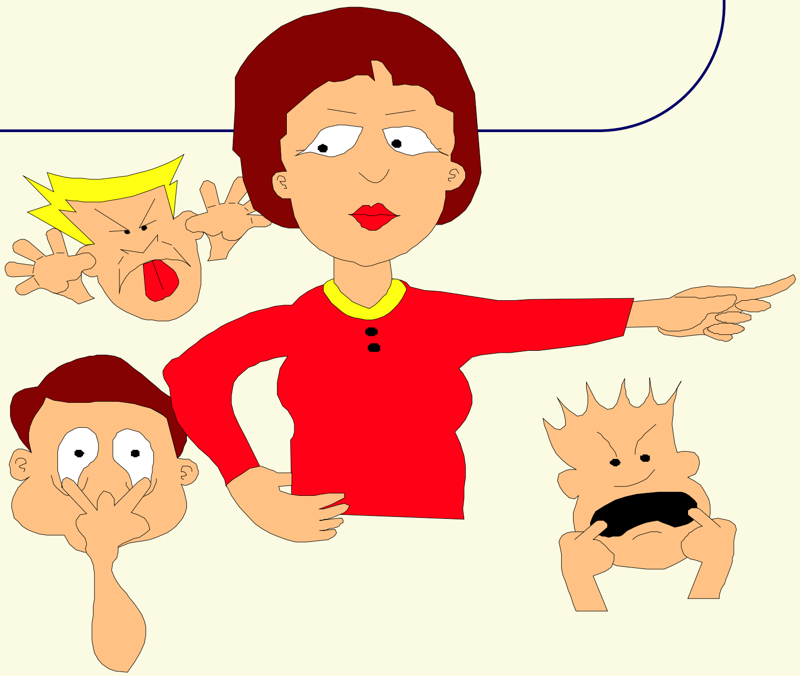
```
package Complex_Numbers is
    type Complex is private;
    function "+" (Left, Right  : Complex) return Complex;
    ... -- similarly "-", "*" and "/"
    function Cartesian_To_Complex (Real, Imag : Float) return Complex;
    function Real_Part (X : Complex) return Float;
    function Imag_Part (X : Complex) return Float;
private
....
end Complex_Numbers;
```

Problem - my package is becoming cluttered and unmanageable. I need to "modularize" it to make it easier to maintain.

Private Children

```
private package Complex_Numbers.Polar is
    procedure Polar_To_Complex (R, Theta : Float) return Complex;
    function "abs" (Right : Complex ) return Float;
    function Arg ( X : Complex) return Float;
end Complex_Numbers.Polar;
```

Solution - create a private child. Only the package body of Complex_Numbers is able to "with Complex_Numbers.Polar". Users of the parent package Complex_Numbers are unable to access any private children.



Golden Rules for Packages

- A specification never needs to *with* its parent; it may *with* a sibling (if compiled first) except that a public child specification may not *with* a private sibling; it may not *with* its own child(it has not been compiled yet
- A body never needs to *with* its parent; it may *with* siblings (private or not); it may *with* its own child.
- A private child is never visible outside the tree rooted at its parent.
- The private part and body of any child can access the private part of its parent (and grandparent...).
- In addition, the visible part of a private child can also access the private part of its parent (and grandparent ...).
- A with clause for a child automatically implies with clauses for all its ancestors.
- A use clause for a unit makes the child units accessible by simple name (this only applies to child units for which there is also a with clause).

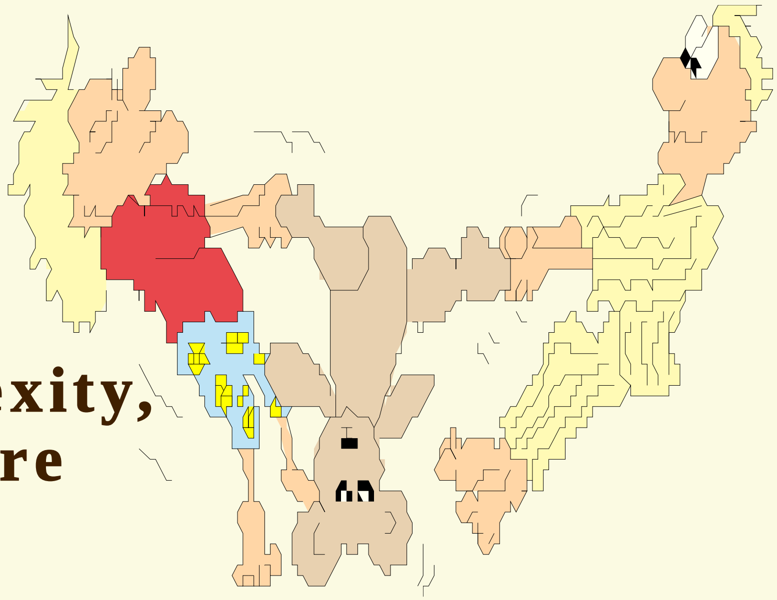
Summary - Child Packages

Public Children allow a large subsystem to be modularized from an user perspective. Users may "with" selected children as needed. Only necessary parts of the larger system need be "withed".

Private Children allow a large system to be functionally decomposed internally. Users of the system are unaware of the modularity.

Both uses reduce recompilation costs.

Both uses reduce complexity, and made the system more manageable.



Summary - OO features of Ada 95

ADTs

- allows non-changeable specification and predefined operations to be shared
- permits method extensions

Tagged types

- used to create parent class for inheritance
- allows code sharing between related classes
- permits attribute & method extensions
- allows class wide variables and dynamic dispatching

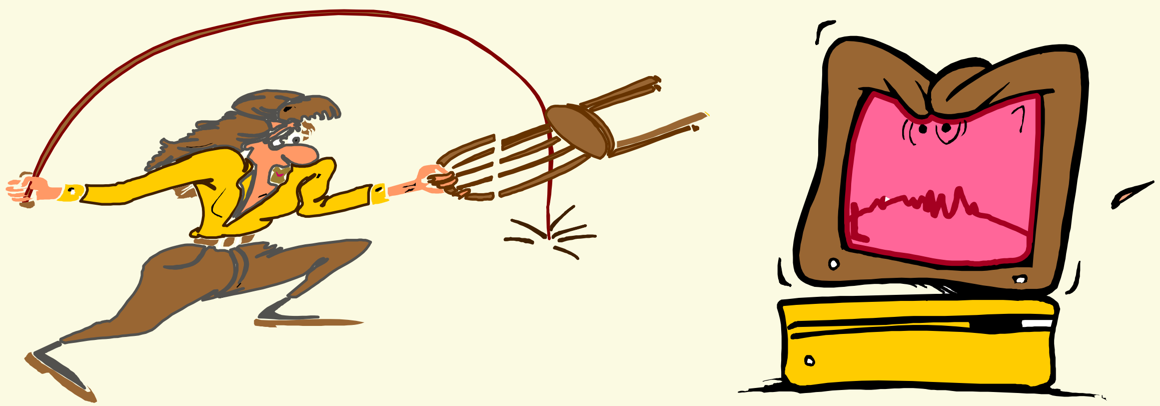
Summary - OO features of Ada 95

Child Packages

- Allows sharing of private views
- Manages namespace
- creates specific views or “schemas” for clients

Multiple Inheritance

- Adding existing properties to existing components



OOSE - Myths and Facts

- Myth - OO can be inserted into programming projects currently under development
- Myth - OO will provide reusable code, thus lowering my overall system cost
- Myth - OO will work with existing methodologies and systems

OOSE - Myths and Facts

- Fact - OO must be part of the overall lifecycle to recognize savings
- Fact - Reuse must be designed into code. It cost more to make reusable code.
- Fact - OO requires supporting CASE tools and extensive training to realize its full benefit

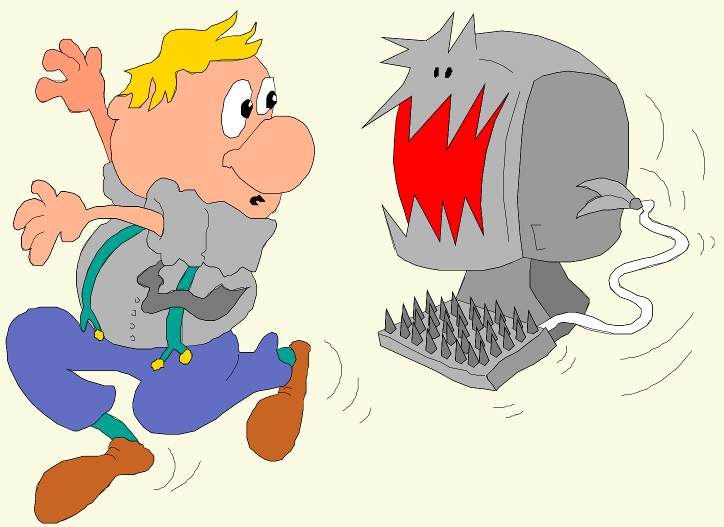
THE BOTTOM LINE

The insertion of OOSE into an organization requires time, money, planning, and commitment!!

Inserting OOSE
Step 1 -Planning and Pre-Insertion Stage

- ✓ Planning - note that Software Process Planning is a KPA (Key Process Area) under Level 2 of the Capability Maturity Model (CMM)
  - Transition Plan
  - Software Development Plan
    - who, what, when, how much, risks

- Changing existing culture
  - Understand the culture change
  - Demonstrate management commitment
    - provide time, tools, and strict enforcement
  - Sell OOSE to those who will use it



Inserting OOSE
Step 2 - Insertion State

- Selecting an OO technique
  - Full life cycle, appropriate domain, target language
- Selecting a CASE tool
  - robust, adequate features
- Staffing and Organizing the project
  - Domain Analyst for reuse, OOP Guru, prototyping expert
- Training the team
  - Adequate time, dedicated time, uncompressed schedule
- Dealing with Legacy Systems
  - reverse-engineer or modify them to fit new OO systems
- Budgeting for Reuse

Inserting OOSE
Step 3 - Project Management

- Analyze, model and prototype
  - Tendency is to over-apply OO for first project
- Effective project tracking and controlling
  - objects, classes
- Define and document the development process
- Collect software metrics
- Inspect OO software products
  - review for quality, develop good metrics, inspect quality of documentation and graphics
- Integrate the documentation

Ada-Predefine Packages

Ada.Exceptions

Ada.Finalization

Numeric, String, Wide String

Control Types

Controlled Types

```
package Ada.Finalization is

type Controlled is abstract tagged private;

 procedure Initialize  (Object: in out Controlled);
procedure Adjust     (Object: in out Controlled);
 procedure Finalize   (Object: in out Controlled);
```

A type derived from Controlled can have an user-defined ***Adjust***, ***Finalize,*** and ***Initialize*** routines. Everytime an object of this type is assigned, released (via exiting scope or freeing up a pointer) or created, the appropriate routine will be called.

Controlled Type Example

```
with Ada.Finalization;

package Bathroom_Stalls is

type Stall is new Ada.Finalization.Controlled with private;

private

  type Stall is new Ada.Finalization.Controlled with
  record
    ID : integer;
  end record;

procedure Initialize  (Object : in out stall);

procedure Adjust    (Object : in out stall);

procedure Finalize   (Object : in out stall);

end Bathroom_Stalls;
```

Controlled Type Example

```
with Ada.Text_IO;
use Ada.Text_IO;
package body Bathroom_Stalls is

Stall_No            : Natural   := 1;
Stalls_In_Use       : Natural   := 0;

procedure Initialize (Object : in out  Stall) is
  begin
    object.id := Stall_No;
    put ("In Initialize, object # " & integer'image(object.id) );
    Stall_No  := Stall_No + 1;
    Stalls_In_Use := Stalls_In_Use + 1;
    Put_Line(".  There are "&integer'image(Stalls_In_Use)
             & " People in stalls.");
end Initialize;

procedure Adjust (Object : in out Stall)
     renames Initialize;

procedure Finalize (Object : in out Stall) is
  begin
    put("In Finalize,   object # " & integer'image(object.id) );
        Stalls_In_Use := Stalls_In_Use - 1;
    Put_Line(".  There are "&integer'image(Stalls_In_Use)
             & " People in stalls.");
end Finalize;

end Bathroom_Stalls;
```

Controlled Type Example

```
with bathroom_stalls;
procedure stalls is

A,B:  bathroom_stalls.stall; --initialize called twice

begin
   declare
   D:  bathroom_stalls.stall; --initialize
   begin
      A := D;   --finalize, then adjust (initialize)
   end;
A := B;         --finalize, then adjust (initialize)
end;
```

```
In Initialize, object #  1.  There are 1 People in stalls.
In Initialize, object #  2.  There are 2 People in stalls.
In Initialize, object #  3.  There are 3 People in stalls.
In Finalize,   object #  1.  There are 2 People in stalls.
In Initialize, object #  4.  There are 3 People in stalls.
In Finalize,   object #  3.  There are 2 People in stalls.
In Finalize,   object #  4.  There are 1 People in stalls.
In Initialize, object #  5.  There are 2 People in stalls.
In Finalize,   object #  2.  There are 1 People in stalls.
In Finalize,   object #  5.  There are 0 People in stalls.
```

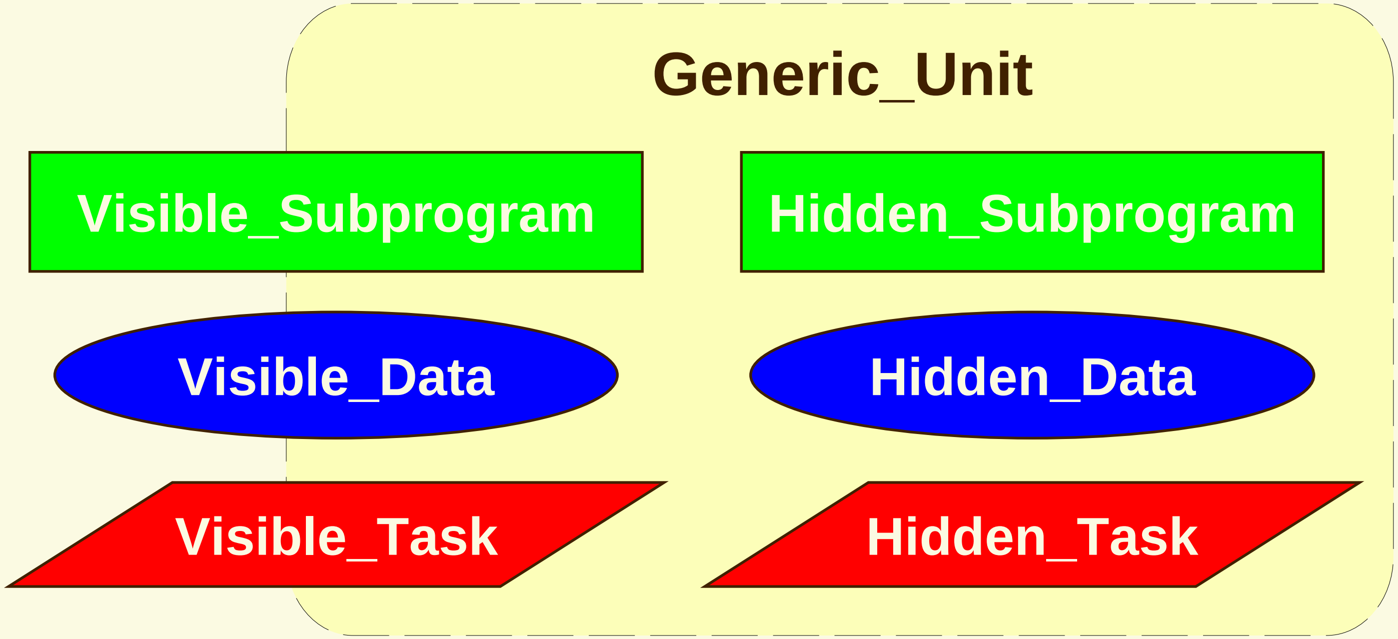
Summary

- Initialize, adjust and finalize executed automatically
- Used when data structures to initialize and finalize with a certain procedure
- User of data structure does not need to call set-up or clean-up procedures

Generics Construction

- Design
- Instantiation
- Generic Parameters:
- Type, Objects, Subprograms, and Packages

Generics

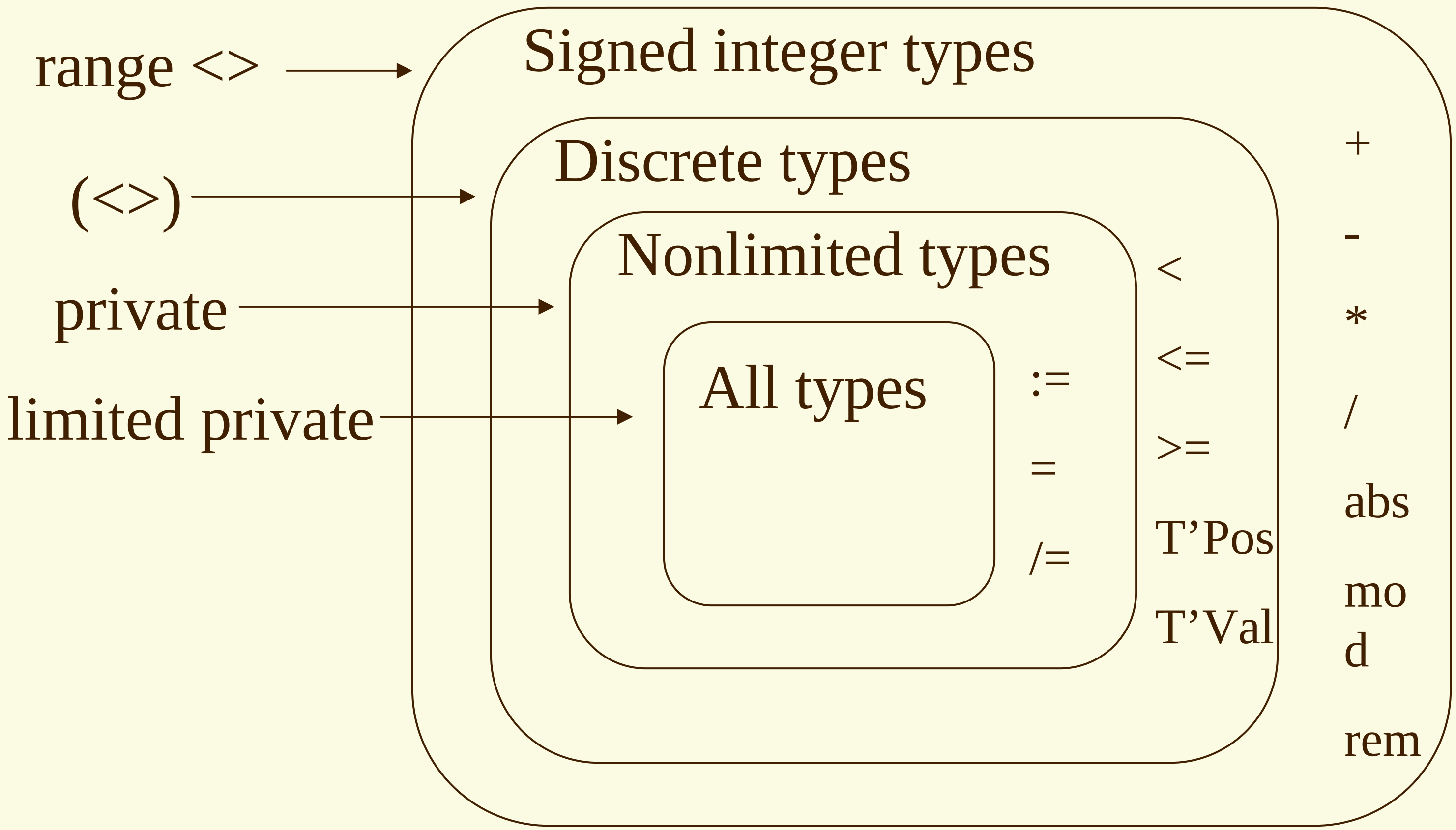


- Template of a subprogram or package
- Used to create actual program units -- “instantiation”
- Promote reusable software
- Improve productivity, maintainability

Generics

- ✓ Provides static polymorphism as opposed to the dynamic polymorphism
- ✓ Static is intrinsically more reliable but usually less flexible.

Generic Types *(Formal parameters)*



Generics

Generic procedure for selection sort

```
-- Generic procedure for selection sort.
generic
  type Index is (<>);
  type Item is private;
  type Vector is array(Index range <>) of 
Item;
  with function "<"(Left, Right: Item) 
return Boolean is <>;
procedure SelectionSort(A: in out Vector);
```

Generics

Generic procedure for selection sort

```
procedure SelectionSort(A: in out Vector) is
  Min: Index;
  Temp: Item;
begin
  for I in A'First .. Index'Pred(A'Last) loop
      Min := I;
      for J in I .. A'Last loop
        if A(J) < A(Min) then Min := J; end if;
      end loop;
      Temp := A(I); A(I) := A(Min); A(Min) := Temp;
  end loop;
end SelectionSort;
```

Generics *generic procedure for selection sort*

```
with SelectionSort;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Float_Text_IO; use Ada.Float_Text_IO;
procedure Sort is
 type Point is
  record
   X, Y: Float;
  end record;
 type  Point_Vector is array(Character range <>) of Point;
 function "<"(Left, Right: Point) return Boolean is
 begin
  return (Left.X < Right.X) or else
       ((Left.X = Right.X) and then (Left.Y < Right.Y));
 end "<";
 procedure Point_Sort is new SelectionSort(
  Character, Point, Point_Vector);
 A: Point_Vector :=
  ((10.0,1.0), (4.0,2.0), (5.0,3.4), (10.0,0.5));
begin
 Point_Sort(A);
 for I in A'Range loop
  Put(A(I).X,5,2,0);
  Put(A(I).Y,5,2,0);
  New_Line;
 end loop;
end Sort;
```

Generics

Generic formal package . . .

```
with Ada.Strings.Bounded;
generic
    with  package Bounded_Length_Instance is
    new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
type String_List_Type is
    array (positive range <>) of
    bounded_Length_Instance.Bounded_String;

procedure Pack_Lines (Lines : in out String_List_Type);
pragma Preelaborate (Pack_Line);
```

Generics

Generic formal package . . .

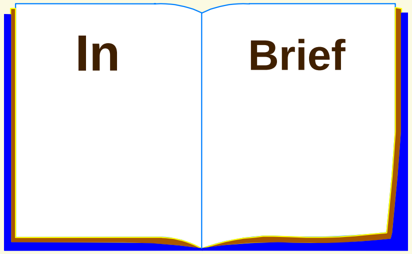
```
with Ada.Strings.Bounded;
package Bounded_80 is
    new Ada.Strings.Bounded.Generic_Bounded_Length
    (Max => 80);
with Bounded_80;
package Bounded_80_Lists  is
    type Bounded_80_List_Type  is
    array (Positive range <>) of
    Bounded_80.Bounded_String;
end Bounded_80_Lists;
```

Generics

Generic formal package . . .

```
with Bounded_80, Bounded_80_Lists, Pack_Lines;
procedure Pack_Lines_80 is
    new Pack_Lines
    (Bounded_Length_Instance => Bounded_80,
     String_List_Type =>
    Bounded_80_Lists.Bounded_80_List_Type);
```

Pack_Line_80 is an ordinary procedure with formal parameter of type bounded_80_Lists.Bounded_80_List_Type.



Summary

- A generic is a form of an Ada program unit that promotes reuse.
- Designing and creating generics is considered an advanced topic of the Ada language.

EXCEPTIONS

Hadnling all posible Erors!

Note : There are four errors in this slide.

Definition

* Ada was developed for real-time systems--systems which should never crash.

EXCEPTIONS ARE

* Events that are infrequent, but not necessarily errors.
    -- Perform some repair action and continue normal execution.

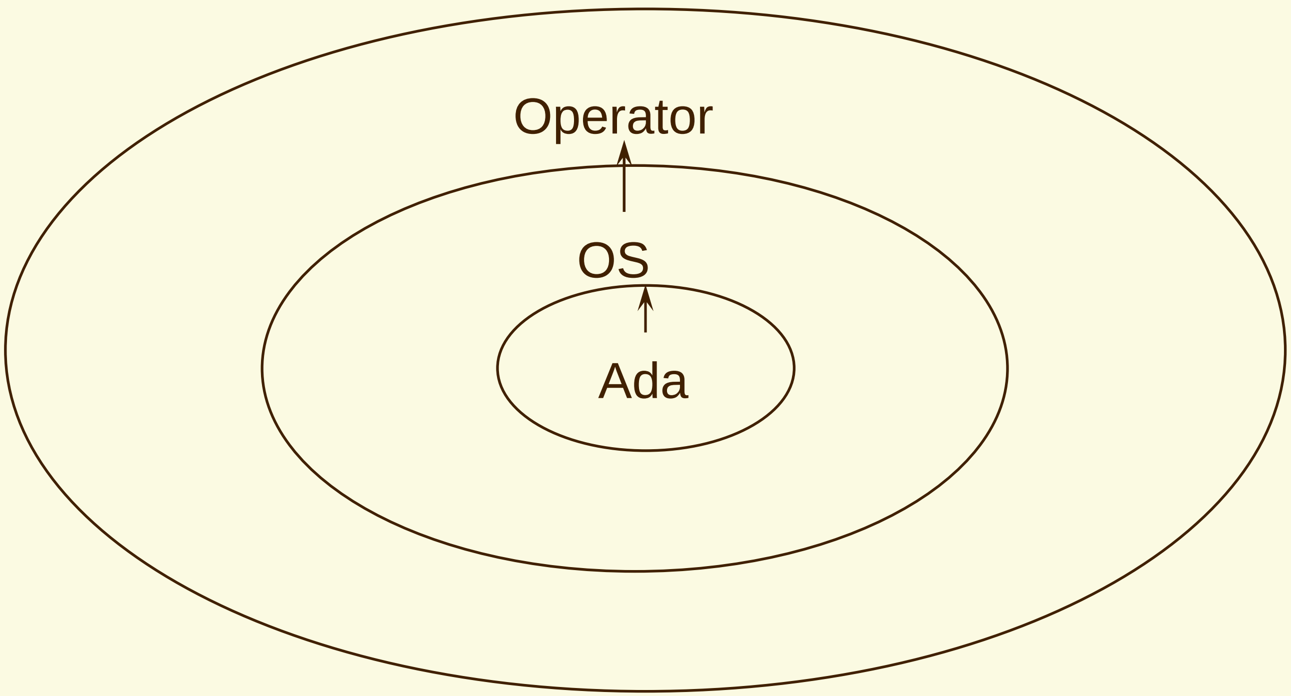
Events that are errors or terminating conditions

-- Execution passes to exception handler but does not return to point where exception was raised.

-- May restart sequence of actions under better conditions

Error Handling Levels

Traditional Approach:



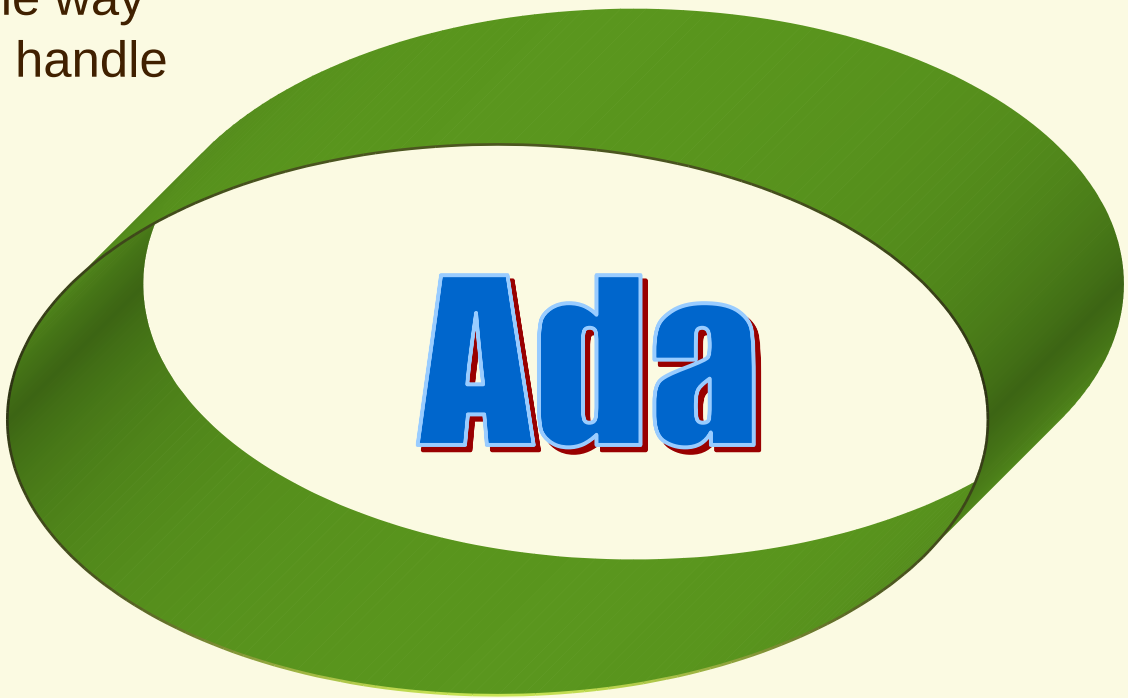
Goals

* To deal with software errors without operator intervention

* To deal with unusual events that are not errors

Exception Handling Levels

This is the way
we try to handle
it in Ada



Example (no exceptions)

```
package STACK_PACKAGE is
        type STACK_TYPE is limited private;
        procedure PUSH (STACK : in out STACK_TYPE;
                        ELEMENT : in ELEMENT_TYPE;
                        OVERFLOW : out BOOLEAN);
end STACK_PACKAGE;

with Ada.Text_IO, STACK_PACKAGE;
use Ada.Text_IO, STACK_PACKAGE;
procedure FLAG_WAVING is
    STACK : STACK_TYPE;
    ELEMENT : ELEMENT_TYPE;
    FLAG  : BOOLEAN;
begin
        PUSH(STACK, ELEMENT, FLAG);
        if FLAG then
              PUT ("Stack overflow");
        end if;
end FLAG_WAVING;
```

Using Exceptions

```
package STACK_PACKAGE is
      STACK_TYPE is limited private;
      STACK_OVERFLOW : exception;
      STACK_UNDERFLOW : exception;
      procedure PUSH (STACK : in out STACK_TYPE;
                             ELEMENT  : ELEMENT_TYPE);
              -- may raise Stack_Overflow
end STACK_PACKAGE;

with Ada.Text_IO, STACK_PACKAGE;
procedure MORE_NATURAL is
      STACK      : STACK_TYPE;
      ELEMENT : ELEMENT_TYPE;
begin
        STACK_PACKAGE.PUSH(STACK, ELEMENT);
exception
      when STACK_OVERFLOW =>
             Ada.Text_IO.PUT ("Stack overflow");
end MORE_NATURAL;
```

Defining Some Terms

* EXCEPTION

  The name attached to a situation that prevents completion of an action. (e.g. CONSTRAINT_ERROR is the name attached to the violation of a constraint).

* RAISING AN EXCEPTION

  Telling the invoker of an action that the corresponding error situation has occurred.

* HANDLING AN EXCEPTION

  Executing some actions in response to this occurrence.

Exceptions

* An exception has a name
  - may be predefined
  - may be declared
* The exception is raised
  - may be raised implicitly by run time system
  - may be raised explicitly by the *raise* statement
* The exception is handled
  - exception handler may be placed in any frame
  - exception propagates until handler is found
  - if no handler anywhere, control is returned to the operating system (like in other systems)

NOTE: A frame is the executable part surrounded by begin - end

Predefined Exceptions

* CONSTRAINT_ERROR

  Violation of range, index, or discriminant constraint...

* NUMERIC_ERROR (obsolete - now CONSTRAINT_ERROR)

  Execution of a predefined numeric operation cannot deliver a correct result

* PROGRAM_ERROR

  Attempt to access a program unit which has not yet been elaborated

* STORAGE_ERROR

  storage allocation is exceeded...

* TASKING_ERROR

  exception arising during inter-task communication

Predefined Exceptions

CONSTRAINT_ERROR:

1) Range constraint violation

   Example:

```
subtype SHORT_INTEGER is
        INTEGER range - 128..127;
COUNTER : SHORT_INTEGER := 128;
```

2) Index constraint violation

   Example:

```
type N_ARRAY is array (1..100) of INTEGER;
An_ARRAY : N_ARRAY;
begin
    AN_ARRAY(0) :=1;
end;
```

3) Discriminant constraint violation

   Example:

```
type P_STRING (LENGTH : POSITIVE := 256) is
record
    DATA : STRING(1 .. LENGTH);
end record;
FILENAME : P_STRING(0);
```

Predefined Exceptions

CONSTRAINT_ERROR:

4) Attempt to use a variant record's component that does not exist

EXAMPLE:

```
type OUTPUT_FORMATS is
        (CHAR_DATA, INTEGER_DATA);

type OUTPUT_DATA (KIND : OUTPUT_FORMATS) is
    record
       case KIND is
          when CHAR_DATA      => CHAR_VALUE : CHARACTER;
          when INTEGER_DATA => INTEGER_VALUE : INTEGER;
       end case;
  end record;

OUTPUT : OUTPUT_DATA (CHARACTER_DATA);

begin
        INT_IO.PUT(OUTPUT.INTEGER_VALUE);
end;
```

5) Attempt to dereference a null pointer

Predefined Exceptions

Numeric_Error (now Constraint_Error):

1) Division by zero

Example:

```
        QUOTIENT, DIVIDEND, DIVISOR : FLOAT;
begin
            DIVIDEND := 1000.0;
            DIVISOR := 0.0;
         QUOTIENT := DIVIDEND / DIVISOR;
end;
```

2) A predefined operation cannot deliver a correct result

Example:

```
        RESULT : INTEGER;
                      -- Assume a 16-bit integer.
            RESULT := 2 ** 32;
```

Predefined Exceptions

3) If a unit is not elaborated before it is referenced.

EXAMPLE:

```
declare
        function CALCULATE (X : INTEGER) return INTEGER;

        X : INTEGER := CALCULATE (2);

        function CALCULATE (X : INTEGER) return INTEGER is
        begin
                return 2**X;
        end CALCULATE;
begin
        null;
end;
```

Predefined Exceptions

Storage_Error:

1) If a collection's storage allocation is exhausted

EXAMPLE:

```
declare
        type PERSONNEL_DATA;
        type DATA_POINTER is
           access PERSONNEL_DATA;
        type PERSONNEL_DATA is
            record
                    NAME : STRING(1 .. 30);
                    NEXT : DATA_POINTER;
            end record;
        FIRST, LAST : DATA_POINTER;
begin
        FIRST := new PERSONEL_DATA;
        LAST := FIRST;
        loop
        LAST.ALL.NEXT := new PERSONNEL_DATA
              LAST := LAST.ALL.NEXT;
        end loop;
end;
```

Predefined Exceptions

2) If storage is not sufficient for elaboration of a declaration or call of a subprogram.

EXAMPLE:

```
procedure Hog is
        type PERSONNEL_DATA
        (LENGTH : POSITIVE := 30) is
        record
                NAME : STRING(1 .. LENGTH);
        end record;
  ME : PERSONNEL_DATA;
  YOU : PERSONNEL_DATA;
begin
        null;
end;
```

3) If a task's storage allocation is exceeded

Predefined Exceptions

TASKING_ERROR :

1) Attempt to rendezvous with a dead task, or

2) Called task is aborted during rendezvous

These predefined exceptions may also be raised by the raise statement.

Raising Exceptions

Three parts of explicitly raising exception:

* Declaring the exception
* Defining the exception handler
* Raising and handling the exception

Declaring Exceptions

exception_declaration ::= identifier_list : exception;

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

Example:

```
procedure Calculation is
    SINGULAR : exception;
    OVERFLOW : exception;
    UNDERFLOW : exception;
begin
-- sequence of statements
end CALCULATION;
```

Example

```
function SUBSTRING(ORIGINAL : STRING;
                   FROM : POSITIVE;
                   SIZE : POSITIVE) return STRING is
   ORIGINAL_LENGTH : POSITIVE;
   INDEX_ERROR : exception;
begin
.
.
.
end SUBSTRING;
```

Exceptions

Three parts to explicitly raised exception:

* Declaring the exception

* Defining the exception handler

* Raising and handling the exception

Defining an Exception Handler

* An exceptional condition is "caught" and "handled" by an exception handler
* Exception handler(s) may appear at the end of any frame (block, subprogram, package or task body)

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

* Form similar to case statement

exception_handler ::=

when exception_choice { |exception_choice} =>sequence_of_statements

exception_choice ::= exception_name | others

Exceptions

The following "frames" may contain handlers.

<u>block stmt</u>	<u>subprogram</u>	<u>package body</u>	<u>task body</u>
declare	**procedure** id **is**	**package body** id **is**	**task body** id **is**
declarations	declarations	declarations	declarations
begin	**begin**	**begin**	**begin**
statements	statements	statements	statements
exception	**exception**	**exception**	**exception**
exception	exception	exception	exception
handlers	handlers	handlers	handlers
end; end;	**end;**	**end;**	

If an exception is raised by the sequence of statements, the frame is searched for a handler for the exception.

Execution will ALWAYS leave the frame, whether or not a handler exists!!

Exceptions

If a handler is found for the exception, then the sequence of statements it contains is executed, and the frame is abandoned.

block statement	→	is finished
subprogram body	→	returns
package body	→	is elaborated
task body	→	is completed

Example

```
with INT_IO, Ada.Text_IO;
procedure MAIN is
    AGE : POSITIVE
begin
    Ada.Text_IO.PUT("Enter age? ");
       INT_IO.GET(AGE);
      -- What happens if characters entered?
end MAIN;

with INT_IO, Ada.Text_IO;
procedure MAIN is
       AGE : POSITIVE;
begin
       Ada.Text_IO.PUT("Enter age? ");
       INT_IO.GET(AGE);
exception
       when Ada.Text_IO.DATA_ERROR =>
         Ada.Text_IO.PUT_LINE("Illegal age.Rerun program.");
end MAIN;
```

Example

```
with INT_IO, Ada.Text_IO, UTILITY_PROGRAMS;
use INT_IO, Ada.Text_IO, UTILITY_PROGRAMS;
procedure RANK_ORDER is
      FIRST, SECOND : TEST_SCORES;
begin
      PUT("Enter first test score");
      GET(FIRST);
      PUT("Enter second test score");
      GET(SECOND);      -- Assume that we input a letter
     if SECOND_IS_GREATER(FIRST, SECOND) then
              SWAP(FIRST, SECOND);
      end if;
end RANK_ORDER;
```

The letter will not get into SECOND. What was in SECOND previously will be in SECOND when the exception is propagated.

Example

```
with INT_IO, Ada.Text_IO, UTILITY_PROGRAMS;
use INT_IO, Ada.Text_IO, UTILITY_PROGRAMS;
procedure RANK_ORDER is
     FIRST, SECOND : TEST_SCORES;
begin
     PUT("Enter first test score");
     GET(FIRST);
     PUT("Enter second test score");
     GET(SECOND);  --  Assume that we input a letter
     if SECOND_IS_GREATER(FIRST, SECOND) then
          SWAP(FIRST, SECOND);
     end if;
exception
     when DATA_ERROR =>
          PUT ("Invalid response--using zero as a default");
          FIRST := 0;
          SECOND := 0;
end RANK_ORDER;
```

NOTE: DATA_ERROR is a predefined exception. (LRM para 14.4)

Restrictions

* Exception handlers must be at the end of a frame

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

* A handler may name any visible exception declared or predefined

* A handler includes a sequence of statements

  - response to exception condition

* A handler for others may be used

  - must be the last handler in the frame

  - handles all exceptions not listed in previous handlers of the frame (including those not in scope of visibility)

  - can be the only handler in the frame

Example

```
with FIX_IT, REPORT_IT, PUNT;
procedure WHATEVER is
      PROBLEM_CONDITION : exception;
begin
      <sequence of statements>
exception
      when PROBLEM_CONDITION =>
            FIX_IT;
      when CONSTRAINT_ERROR =>
            REPORT_IT;
      when others =>
            PUNT;
end WHATEVER;
```

Exceptions

Three parts to explicitly raised exception:

* Declaring the exception
* Defining the exception handler
* Raising and handling the exception

How Exceptions Are Raised

* Implicitly by run time system

  - predefined exceptions

Explicitly by raise statement

raise_statement ::=raise [exception_name];

-the name of the exception must be visible at the point of the raise statement

-a raise statement without an exception name is allowed only within an exception handler

Implicitly Raised Exceptions

```
function TOMORROW (DAY : DAYS_OF_THE WEEK)
    return DAYS_OF_THE_WEEK is
begin
    return DAYS_OF_THE_WEEK'SUCC(DAY);
exception
    when CONSTRAINT_ERROR =>
        return DAYS_OF_THE_WEEK'FIRST;
end TOMORROW;
```

frame

Explicitly Raised Exceptions

```
function TOMORROW (DAY:DAYS) is
      LAST_DAY : exception;
begin
      if DAY = DAYS'LAST then
            raise LAST_DAY;
      else
            return DAYS'SUCC(DAY);
      end if;
exception
      when LAST_DAY =>
            return DAYS'FIRST;
end TOMORROW;
```

Example

```
procedure WHATEVER is
    PROBLEM_CONDITION : exception;
    REAL_BAD_CONDITION : exception;
begin
    if PROBLEM_ARISES then
        raise PROBLEM_CONDITION;
    end if;
    if SERIOUS_CONDITION then
        raise REAL_BAD_CONDITION;
    end if;

exception
    when PROBLEM_CONDITION =>
        FIX_IT;
    when CONSTRAINT_ERROR =>
        REPORT_IT;
    when others =>
        PUNT;
end WHATEVER;
```

Effects of Raising an Exception

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

(2) Exception is lowered

(3) Sequence of statements in exception handler is executed

(4) Control passes to end of frame

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

Exceptions

how can the same exception be raised again?

* Within a handler, the exception that caused transfer to the handler may be raised again by a normal raise statement (mentioning its name) or by a raise statement of the form

```
raise;
```

* Abandons the execution of the frame and propagates the corresponding exception.

Example

```
with Ada.Text_IO, MATRIX_OPS;
procedure EXCEPTIONAL is
      SINGULAR : exception
begin
      if MATRIX_OPS.MATRIX_IS_SINGULAR then
                raise SINGULAR
      end if;
      --        sequence of statements
exception
      when SINGULAR =>
              Ada.Text_IO.PUT("Matrix is singular"):
      when others =>
              Ada.Text_IO.PUT("Fatal error");
                raise;
end EXCEPTIONAL;
```

Example - No Exception handling

Example with no exception handling:

```
with INT_IO, Ada.Text_IO;
procedure MAIN is
    AGE : POSITIVE;
begin -- main
    Ada.Text_IO.PUT("Enter Age:");
    INT_IO.GET(AGE);
end;
```

Example with no recovery

```
with INT_IO, Ada.Text_IO;
procedure MAIN is
    AGE : POSITIVE;
begin
    Ada.Text_IO.PUT("Enter_Age:");
    INT_IO.GET(AGE);
exception
    when Ada.Text_IO.DATA_ERROR =>
            Ada.Text_IO.PUT("Illegal Age");
end MAIN;
```

Example with recovery

```
with INT_IO, Ada.Text_IO;
procedure MAIN is
    AGE : POSITIVE;
begin
    Ada.Text_IO.PUT("Enter Age:");
    loop
        begin  -- A new block
            INT_IO.GET(AGE);
            exit; -- exit the loop if Age is OK
        exception
            when Ada.Text_IO.DATA_ERROR =>
              Ada.Text_IO.PUT_LINE(
                    "Illegal age. Reenter:");
        end;
    end loop;
end MAIN;
```

Example - restricted number of retries

```
with INT_IO, Ada.Text_IO;
procedure MAIN is
    AGE : POSITIVE;
    NUMBER_OF_RETRIES : NATURAL := 0;
    TOO_MANY_MISTAKES : exception;
begin
    Ada.Text_IO.PUT("Enter Age:");
    loop
        begin
            INT_IO.GET(AGE);
            exit; -- exit the loop if Age is OK
        exception
            when Ada.Text_IO.DATA_ERROR =>
                if NUMBER_OF_RETRIES = 10 then
                    raise TOO_MANY_MISTAKES;
                end if;
                NUMBER_OF_RETRIES := NUMBER_OF_RETRIES + 1;
                Ada.Text_IO.PUT_LINE ("Illegal age.Reenter:");
        end;
    end loop;
exception
    when TOO_MANY_MISTAKES =>
        Ada.Text_IO.PUT_LINE ("Too many mistakes made.);
end MAIN;
```

Exceptions

Exceptions may be raised by the elaboration of declarations. Then, the corresponding frame is NOT searched for a handler.

```
with Ada.Text_IO;
procedure BAD_STRING is
    NAME : STRING (0 .. 20);  -- Needs to begin with 1
begin
    null;
exception
    when others =>
        Ada.Text_IO.PUT_LINE("Exiting main program.");
end BAD_STRING;
```

NOTE- the above program will terminate, but the exception handler will not be invoked.

Catching declaration exceptions

```
with Ada.Text_IO;
procedure BAD_POSITIVE is
begin
        declare
                VALUE : POSITIVE := -1;
        begin
                null;
        end;
exception
        when others =>
                Ada.Text_IO.PUT_LINE("Exiting main program.");
end BAD_POSITIVE;
```

No recovery if error

```
with SEQUENTIAL_IO;
use SEQUENTIAL_IO;
procedure OPERATE (NAME : STRING) is
    FILE : FILE_TYPE;
begin
    OPEN(FILE, INOUT_FILE, NAME);

    -- Process the file (This could "bomb" and leave the  file open.)

    CLOSE(FILE);
end OPERATE;
```

Giving code its' Last Wishes

```
with SEQUENTIAL_IO;
use SEQUENTIAL_IO;
procedure SAFE_OPERATE(NAME : in STRING) is
        FILE : FILE_TYPE;
begin
        OPEN(FILE, INOUT_FILE, NAME);

        -- Operate on the file (may raise an exception)

        CLOSE(FILE);
exception
        when others =>
         CLOSE(FILE);
                raise;
end SAFE_OPERATE;
```

Exceptions

* When an exception is raised within the sequence of statements of a frame, the execution of this sequence of statements is always abandoned.

What happens next depends on the presence or absence of appropriate exception handlers.

-- Subprogram body =>

Same exception is raised implicitly at the point of call of the subprogram (except for a main program).

Absence

-- Block statement => between "begin" and "end"

Same exception is raised within the frame containing the block statement.

* In either case we say that the exception is "propagated"

How Exceptions Can Be Useful

* Normal processing could continue if
  - the cause of exception condition can be "repaired"
  - an alternative approach can be used
  - the operation can be retired

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

* If termination is necessary, "clean-up" can be done first

Example

```
-- With no exception handlers, control passes to
-- surrounding frame.
with Ada.Text_IO;
procedure GET_NUMBERS is
    type NUMBERS is range 1 .. 100;
    package NUM_IO is new
        Ada.Text_IO.INTEGER_IO (NUMBERS);
    A_NUMBER : NUMBERS;
begin
    loop
        NUM_IO.GET (A_NUMBER);
        Ada.Text_IO.NEW_LINE;
        Ada.Text_IO.PUT("The number is");
        NUM_IO.PUT(A_NUMBER);
        Ada.Text_IO.NEW_LINE;
    end loop;
end GET_NUMBERS;
```

Example

```
-- With an exception handler, you retain control.
with Ada.Text_IO;
procedure GET_NUMBERS is
    type NUMBERS is range 1 .. 100;
    package NUM_IO is new
        Ada.Text_IO.INTEGER_IO (NUMBERS);
    A_NUMBER : NUMBERS;
begin
    loop
        NUM_IO.GET (A_NUMBER);
        Ada.Text_IO.NEW_LINE;
        Ada.Text_IO.PUT("The number is");
        NUM_IO.PUT(A_NUMBER);
        Ada.Text_IO.NEW_LINE;
    end loop;
exception
    when Ada.Text_IO.DATA_ERROR =>
        Ada.Text_IO.SKIP_LINE;
        Ada.Text_IO.PUT_LINE ("That was a bad
    number");
end GET_NUMBERS;
```

Example

```
-- Unrestricted number of retries

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

Package Ada.Exception

```
        package Ada.Exceptions is
        type Exception_Id is private;
        Null_Id : constant Exception_Id;
        function Exception_Name(Id : Exception_Id) return String;
                type Exception_Occurrence is limited private;
        type Exception_Occurrence_Access is access all Exception_Occurrence;
        Null_Occurrence : constant Exception_Occurrence;
                procedure Raise_Exception(E : in Exception_Id; Message : in String := "");
        function Exception_Message(X : Exception_Occurrence) return String;
        procedure Reraise_Occurrence(X : in Exception_Occurrence);

                function Exception_Identity(X : Exception_Occurrence) return
Exception_Id;
        function Exception_Name(X : Exception_Occurrence) return String;
                -- Same as Exception_Name(Exception_Identity(X)).
        function Exception_Information(X : Exception_Occurrence) return String;
                procedure Save_Occurrence(Target : out Exception_Occurrence;
                   Source : in Exception_Occurrence);
        function Save_Occurrence(Source : Exception_Occurrence)
                  return Exception_Occurrence_Access;
private
        ... -- not specified by the language
```

Example Package Ada.Exception

```
With Ada.execptions;
Q : aliased Queue;
Ex1, Ex2, Ex3, Ex4 : exception;
procedure P1 is
begin
Ada.Exceptions.Raise.Exception (Ex1'identity,"P1" & integer'image(13));
exception
when E: others => Put((13,Save.Occurrence(E), Q);
end P1;
…….
End Test;
```

Example Package Ada.Exception

Retrieved exception occurrences

```
…...
begin
P1;
while not Empty(Q) loop
begin
Ada.Exceptions.ReRaise.Occurrence(Get(Q'Access).Occurrence.all);
exception
when E: others => Put_Line(Exception_Information(E);
end;
end loop;
end Test_Retrieve;
```

Summary

* Exceptions are events considered as errors or terminating conditions
* Execution of the current frame is always abandoned and control passes to an exception handler
* The handler may restart sequence of statements under better conditions by nesting frames in a loop