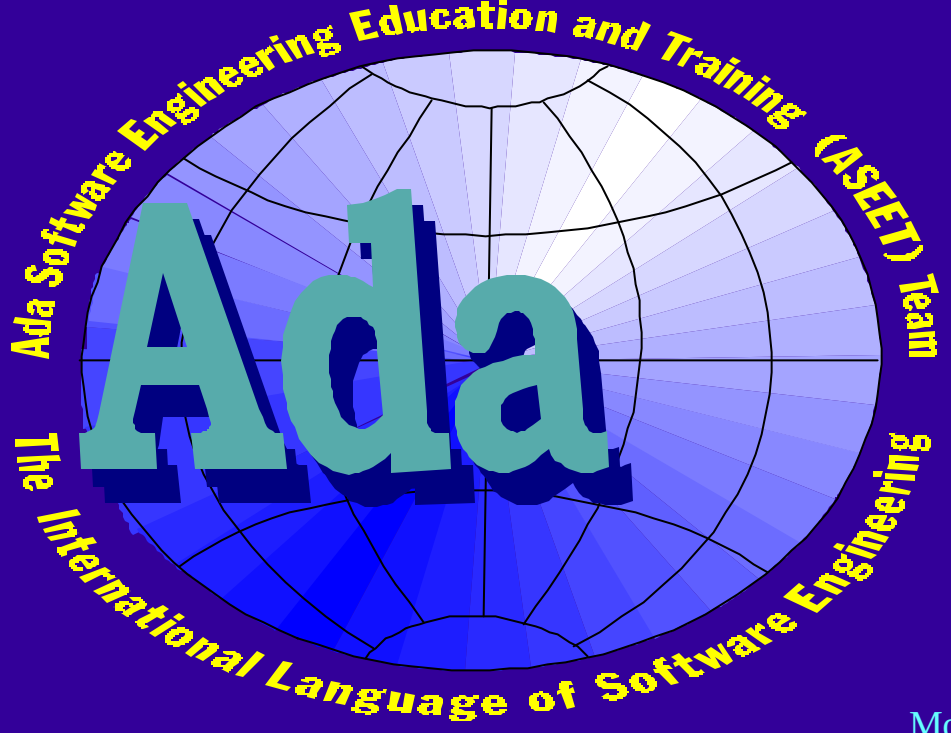
Introduction to Ada

ANSI/ISO/IEC-8652:1995

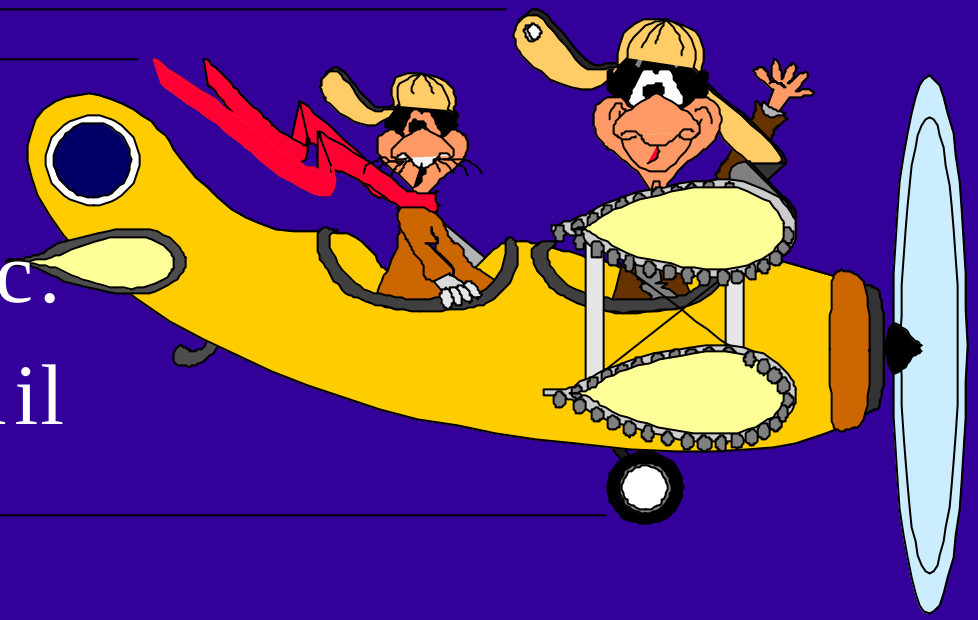


Speakers

Eugene Bingue, Ph.D.
Dr.Bingue@ix.netcom.com

David A. Cook, Ph.D.
USAF STSC/Shim Inc.
David.Cook@hill.af.mil

Leslie (Les) Dupaix
USAF STSC
Les.Dupaix@hill.af.mil

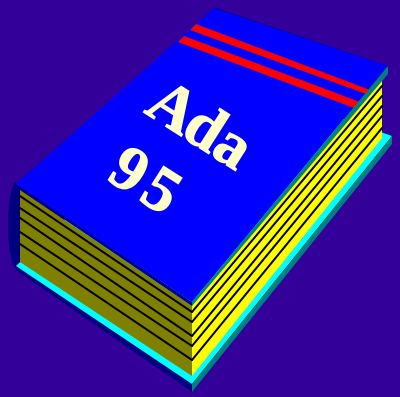


This part of the tutorial will....

- Give you a “quick and dirty” “firehose in the mouth” approach to the core language of Ada 95. It presents the basic language, similar to the Ada 83 language
- Discuss the building blocks of Ada programs, and discuss how packaging and strong typing provide for safe and reliable code.

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

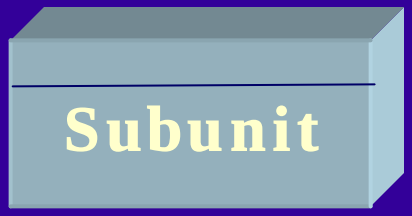
Strong typing

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

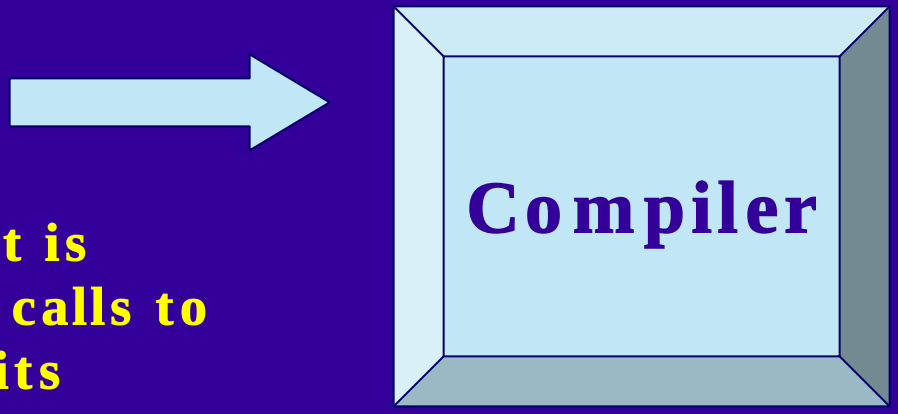
```
procedure TESTER is
    type APPLES is range 0 .. 100; -- an integer type
    type ORANGES is range 0 .. 100; --ditto
    A : APPLES := 100;              - - initialize during declaration
    O : ORANGES;
begin
    O := A;                    - -  ERROR, incompatible types
end;
```

--> A and O are not compatible.

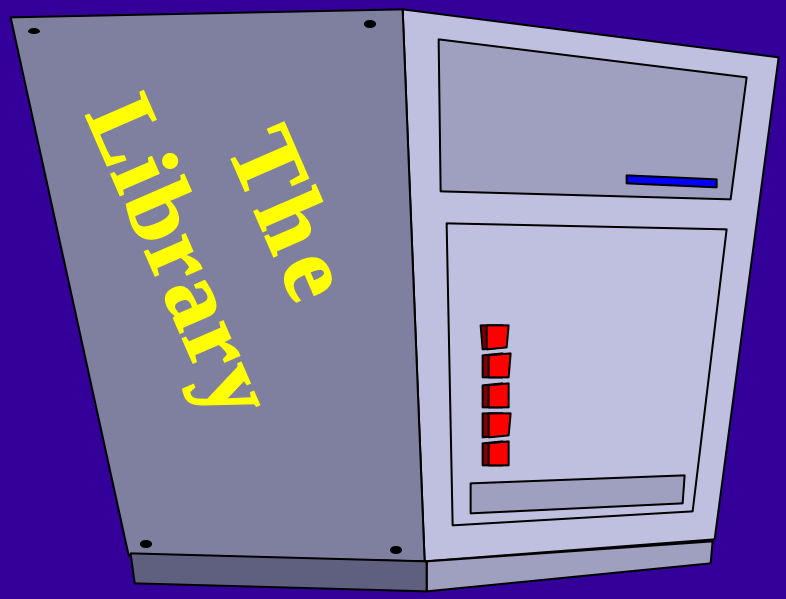
The Library



The subunit is developed with calls to library units



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



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

Assignment Statements

VARIABLE := EXPRESSION ;

* The variable takes on the value of the expression

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

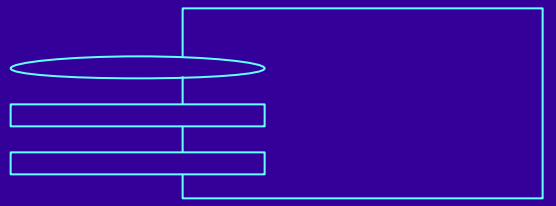
```
MY_INT := 17;                    -- Integer

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

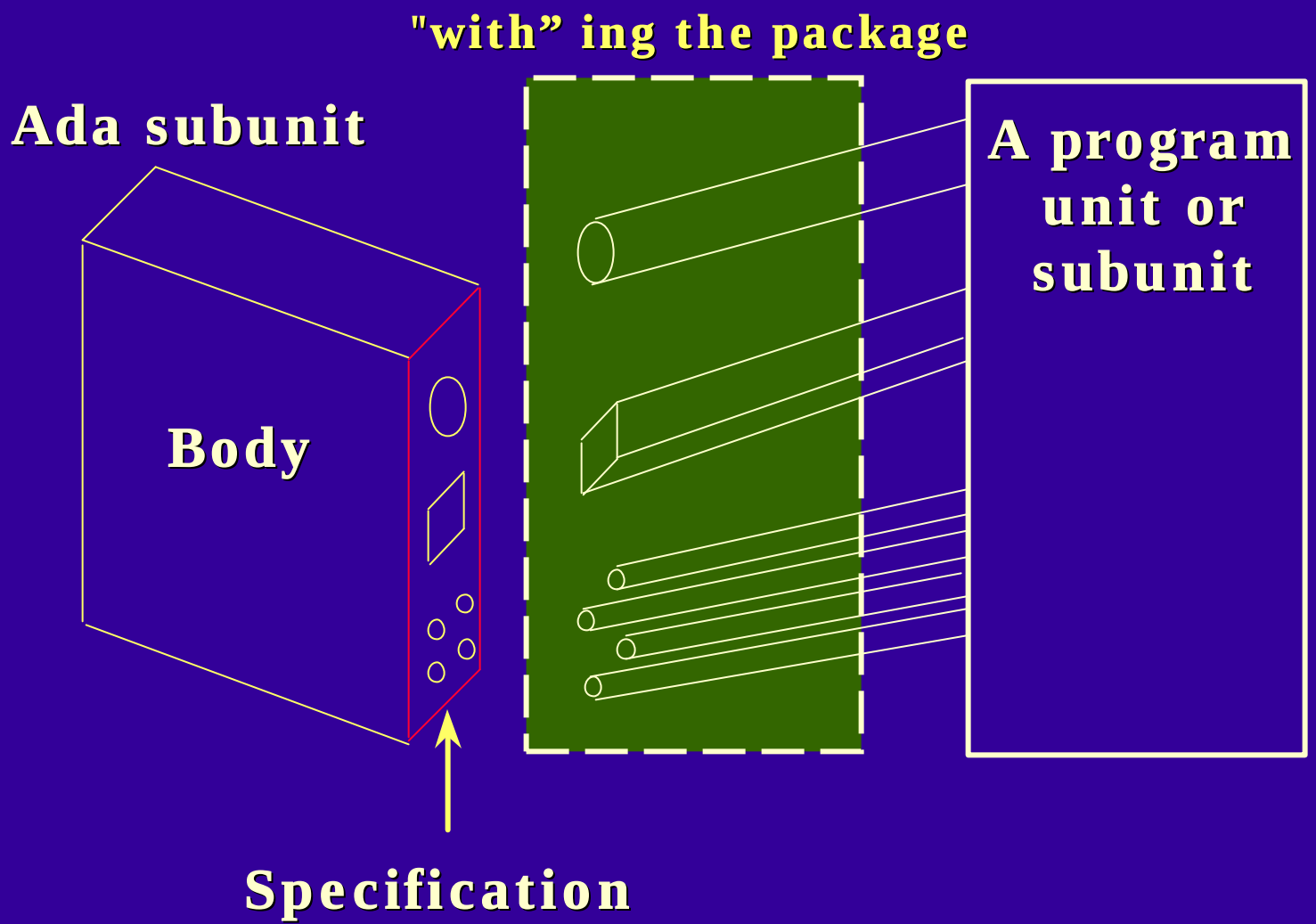
Tasks
- -- Parallel Processing
- -- Real-time Programming
- -- Interrupt Handling

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



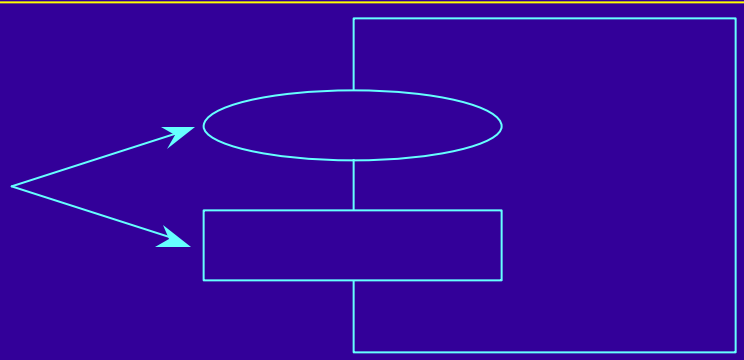
Generics
- -- Templates

Specification and Bodies



Subprogram Units

Specification



Body

"What" the program unit does

← **ABSTRACTION** →

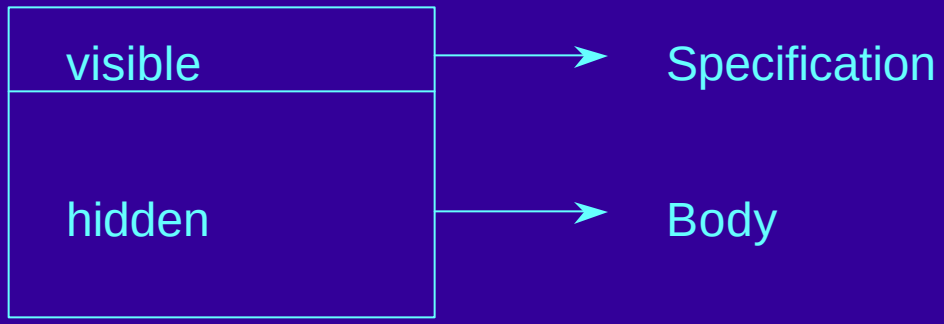
"How" the program unit does what it does

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

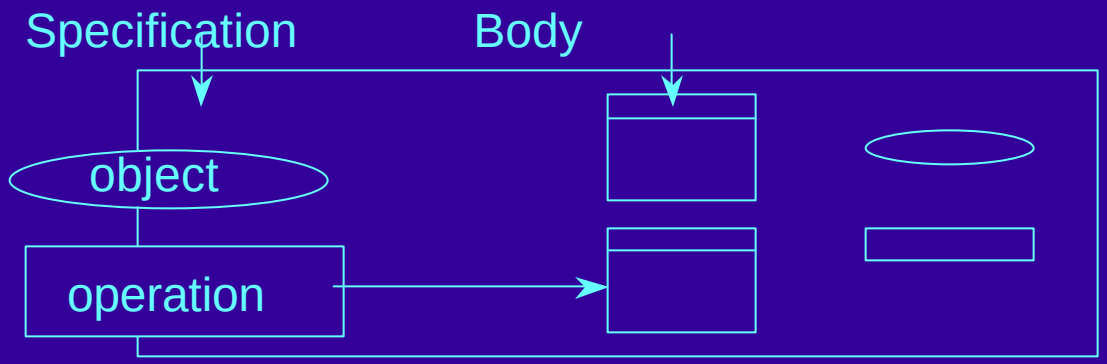
```
-- Procedure Body
procedure SWAP (PRE, POST: in out INTEGER) is
   TEMP: INTEGER := PRE;
begin
   PRE        := POST;
   POST       := TEMP;
end SWAP;
```

- 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.

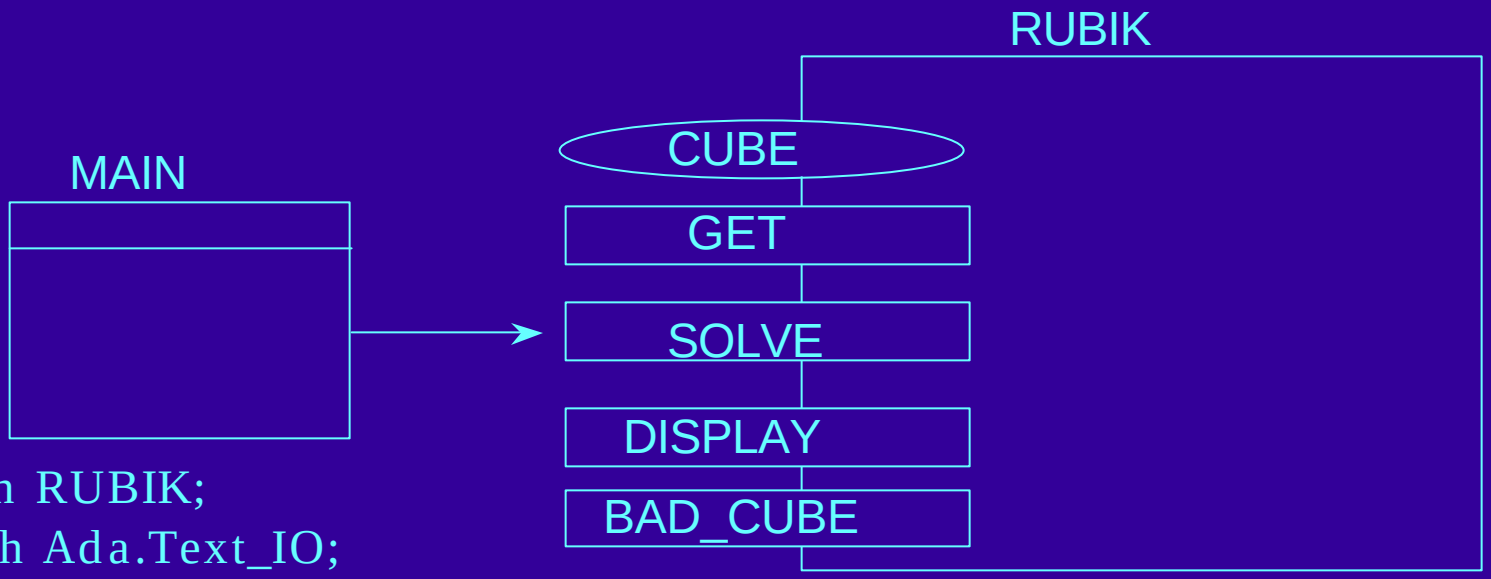
* Operations (subprograms, functions, etc) whose specification appear in the package specification must have their body appear in the package body.

* 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;
```

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

Ada Tasks

- Each task is a separately executing unit (with it's own stack)
- A task starts running as soon as the parent starts executing
- Once a task starts, it runs independently of the parent
- If desired, a task can communicate with other running units by passing data. Data is passed via "Entry Points" using rendezvous (calls)

Task Communication

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

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

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

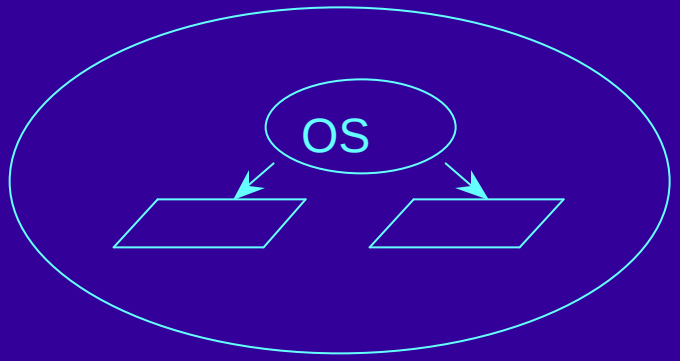
Tasking Example

task specs

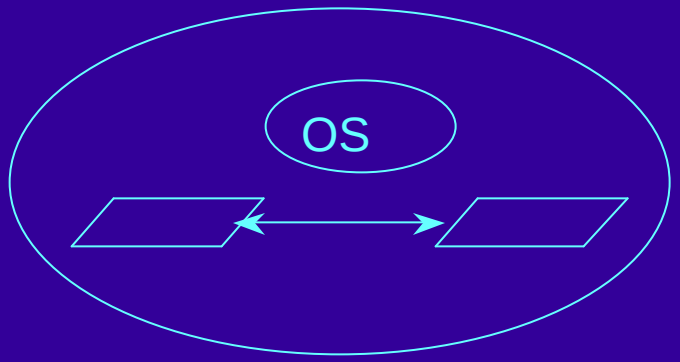
task bodies

```
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;
```

Tasks are Operating System Independent

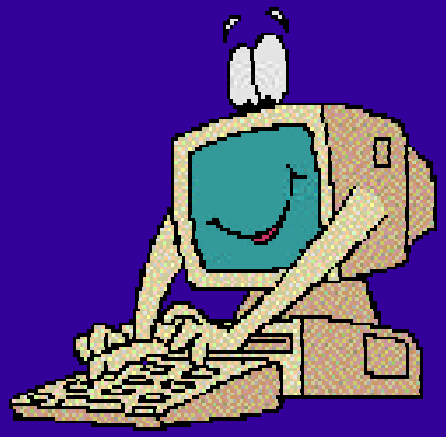


THE TRADITIONAL MODEL OF CONCURRENCY



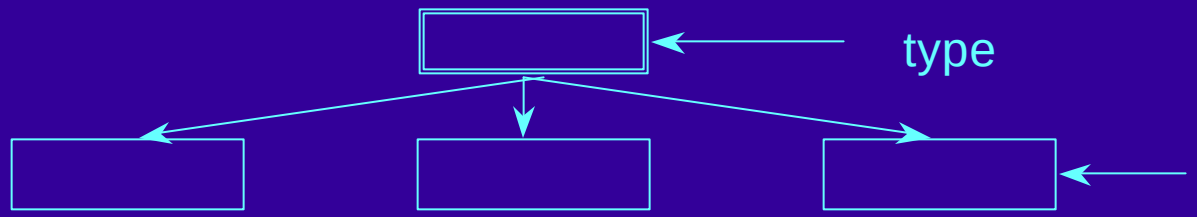
The Ada TASKING MODEL

Typing



Ada Types

- An Ada type is a template for objects
  - a set of values which are meaningful
  - a set of operations on the objects (values)
- All objects must be declared and objects of different types cannot be implicitly mixed in operations
- TYPES are not operated upon directly, objects can be operated upon. Types are a means of declaring OBJECTS (instances of the type).



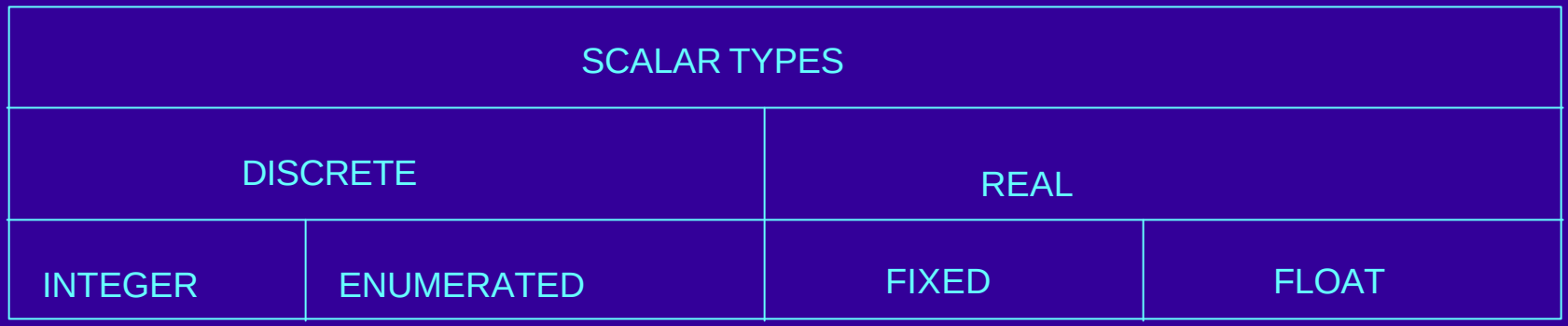
Classes of Ada Types

- **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

Predefined Types

- In Ada, most types SHOULD be declared by the user
  - Models real-world
  - Has useful limits
  - Matches the “abstraction” being programmed
- Predefined types also exist

Predefined Ada types



Integer
natural
positive
long_integer
short_integer

boolean
character

duration
decimal

float

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;
```

ELEMENT OBJECT DECLARATIONS

```
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 Type

CHARACTER TYPE DECLARATIONS -- based upon ISO 8859-1

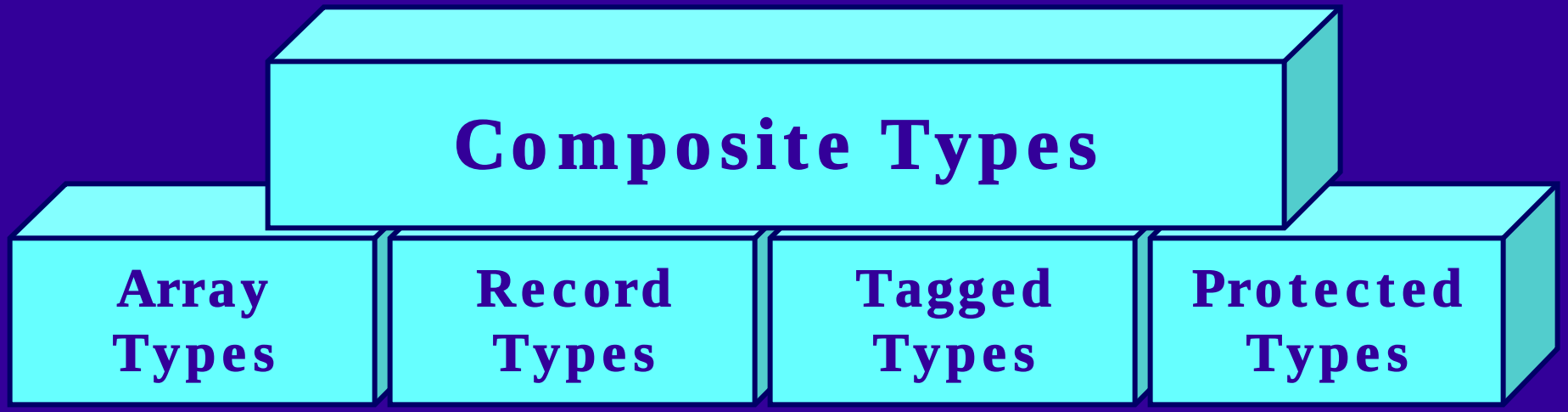
```
type ROMAN_DIGIT is ('I', 'V', 'X', 'L', 'C', 'D', 'M');

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

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

CHARACTER OBJECT DECLARATIONS

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

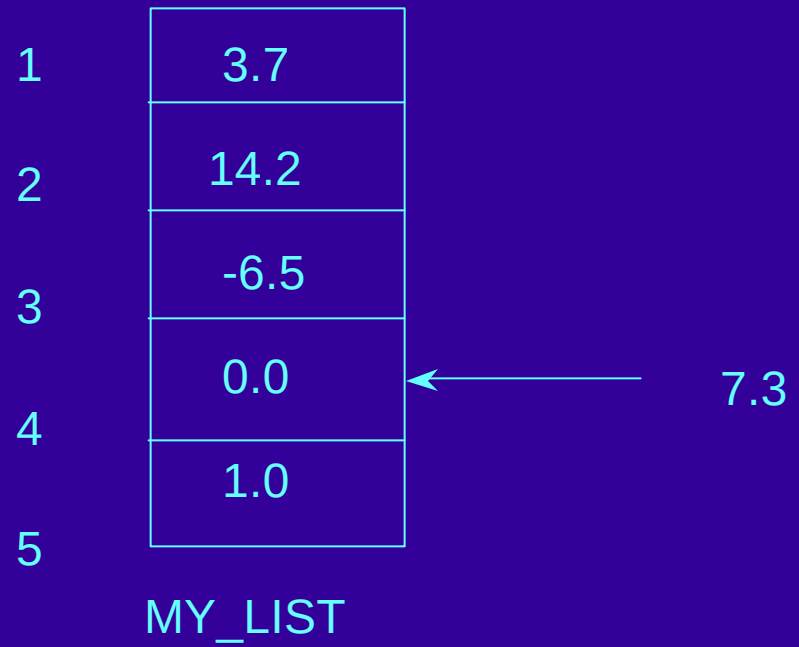
Composite Types

- Combination of any type
- Building block for complex data structures



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;
```

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

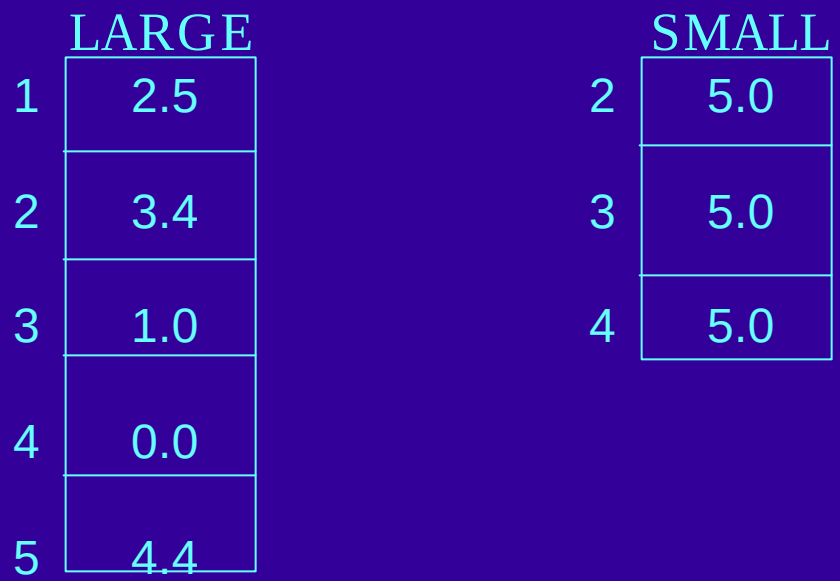
MY_WEEK

```
if MY _WEEK (THU) = TRUE then ...
if MY_WEEK (THU) then
```

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);
```



Record Type

RECORD TYPE DECLARATION

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

        end record;
```

RECORD OBJECT DECLARATION

```
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 := 1965;
                                        --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 ...
```

Ada 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

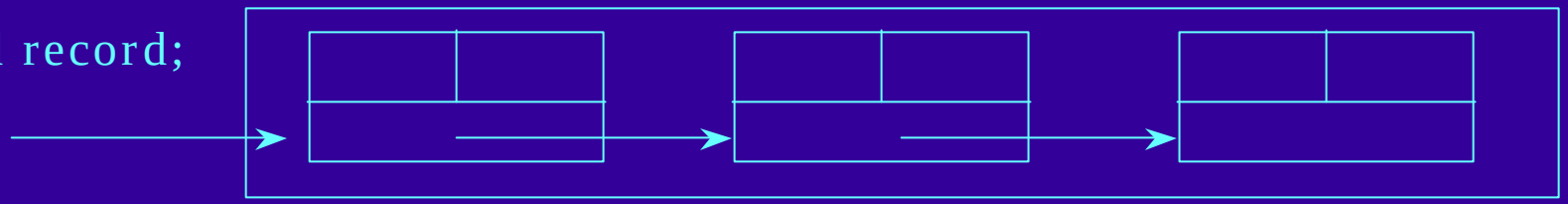
```
subtype TEN_LONG is STRING (1 .. 10);
FIRST_TEN : TEN_LONG := "HEADER     ";
```

FIRST_TEN

H	E	A	D	E	R				

Access Types
("pointers", but don't tell anyone)

```
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
- Two types
  - Private -- assignment, (in) equality and all explicitly declared operations
  - Limited Private -- assignment and all explicitly declared operations

Private types

```
Package PASSWORD_CHECK is
    type PASSWORD_TYPE is limited private;
    procedure GET_PASSWORD ( Value : in out PASSWORD_TYPE);
    procedure VERIFY_PASSWORD ( Value : in out PASSWORD_TYPE);
private
    type PASSWORD_TYPE is STRING(1..20);
end PASSWORD_CHECK;
```

- Password is “hidden” in that the compiler will enforce limitations on what you can do with any variable of type PASSWORD_TYPE
- If you “with PASSWORD_CHECK” in another unit, you have access to PASSWORD_TYPE and the two procedures, but you may NOT perform ANY other operations on a variable of type PASSWORD_TYPE

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, ...)

Conditional Statements (IF)

```
if TODAY = (30, JUL, 1943) then
    PEGS_YEARS := PEGS_YEARS + 1;
    GET (BIRTHDAY_CARD);
end if;

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;
```

Conditional Statements (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;
   when 9 .. 20   => DO_SOMETHING RADICAL;
   when others    => PUNT;
end case;
```

Iteration Statements (loop)

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


for i in 1..5 loop
   GET_IT(NUMBER);
    MODIFY_IT (NUMBER);
end loop;


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

What makes Ada loops unique

- You can only increment or decrement (no skipping values)
- The LCV (Loop Control Variable) is IMPLICITELY declared inside the loop
- The LCV disappears when the loop terminates

Sample Loop

```
....
COUNTER : Integer := 0;
...
for Counter in 1..100 loop
    if Array(Counter) = SOME_VALUE then exit; end if;
end loop;
put (Counter);   - -  Prints out the value of Counter
```

QUESTION - what is the value of Counter for the Put operation?

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;
```

Sample Program

```
function WRAP ( D : DAYS ) return DAYS is
begin
    if D = DAYS'LAST then
          return DAYS'FIRST
    else
          return  DAYS'SUCC(D);
    end if;
end WRAP;
```

EXCEPTIONS

Hadnling all posible Erors!

Note : There are four errors in this slide.

Exceptions

- Exceptions are a mechanism to allow the programmer to identify and handle errors within the program without calling system error routines.
- There are two kinds of exceptions:
  - predefined exceptions
    - Standard (Constraint, Program, Storage, Tasking)
    - IO (Status, Mode, Name, Data, Layout, End, Device, Use)
  - user-defined exceptions
    - Allows the designer to define and raise non-standard exceptions

Exceptions

- *Exception handlers* handle the exception.
  - You can also *name* and *save* an exception externally.
  - An exception can be *re-raised* and it can be *propagated*.

Exception Example

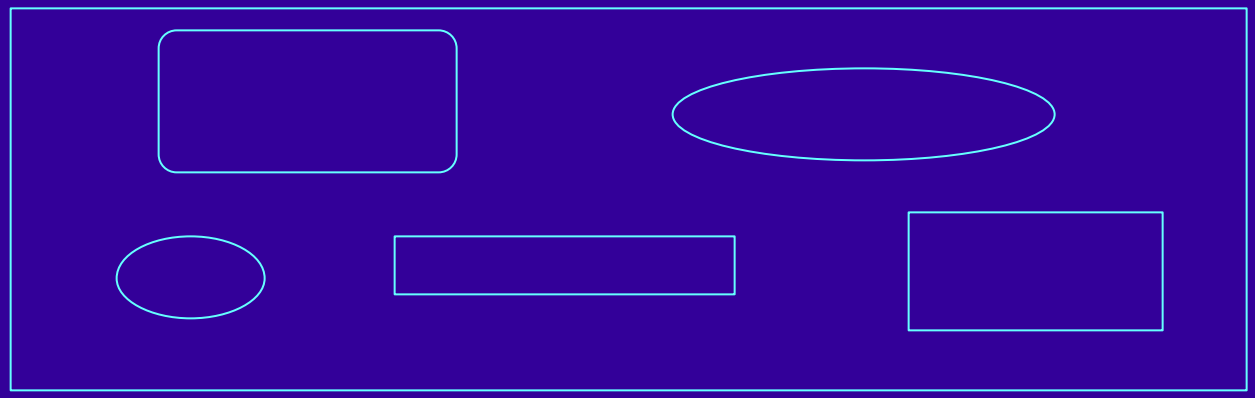
```
My_error : exception;
AGE : Positive;
DATA : string (1..10) := (others => ' ');
begin
        Get_AGE (Age);
        Get_DATA (DATA);
        If DATA = "David Cook" then
                raise my_error;
        end if;
Exception
        when constant_error =>   ....
        when my_error =>  ....
End;
```

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

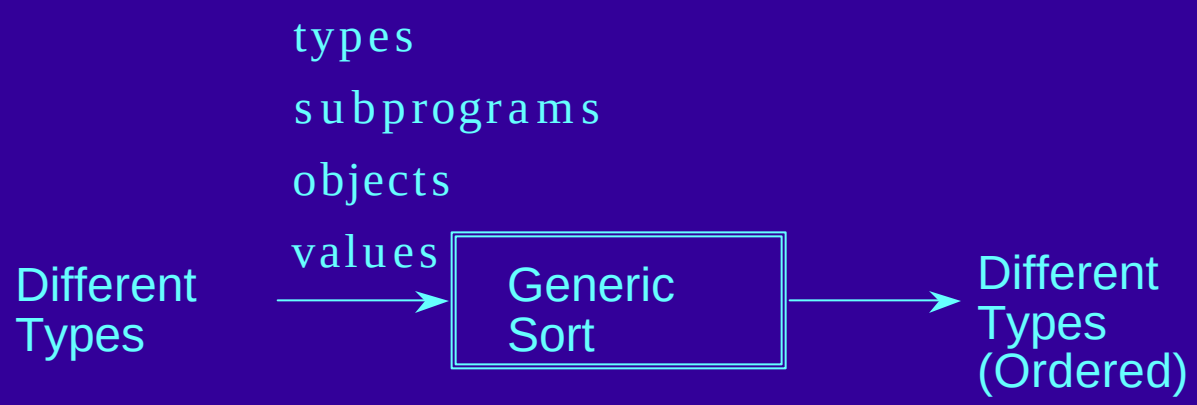
* Defines a template for a general purpose program units

* The template must be instantiated prior to use in a program

* Strong typing makes generics a necessary language feature

Reason for 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

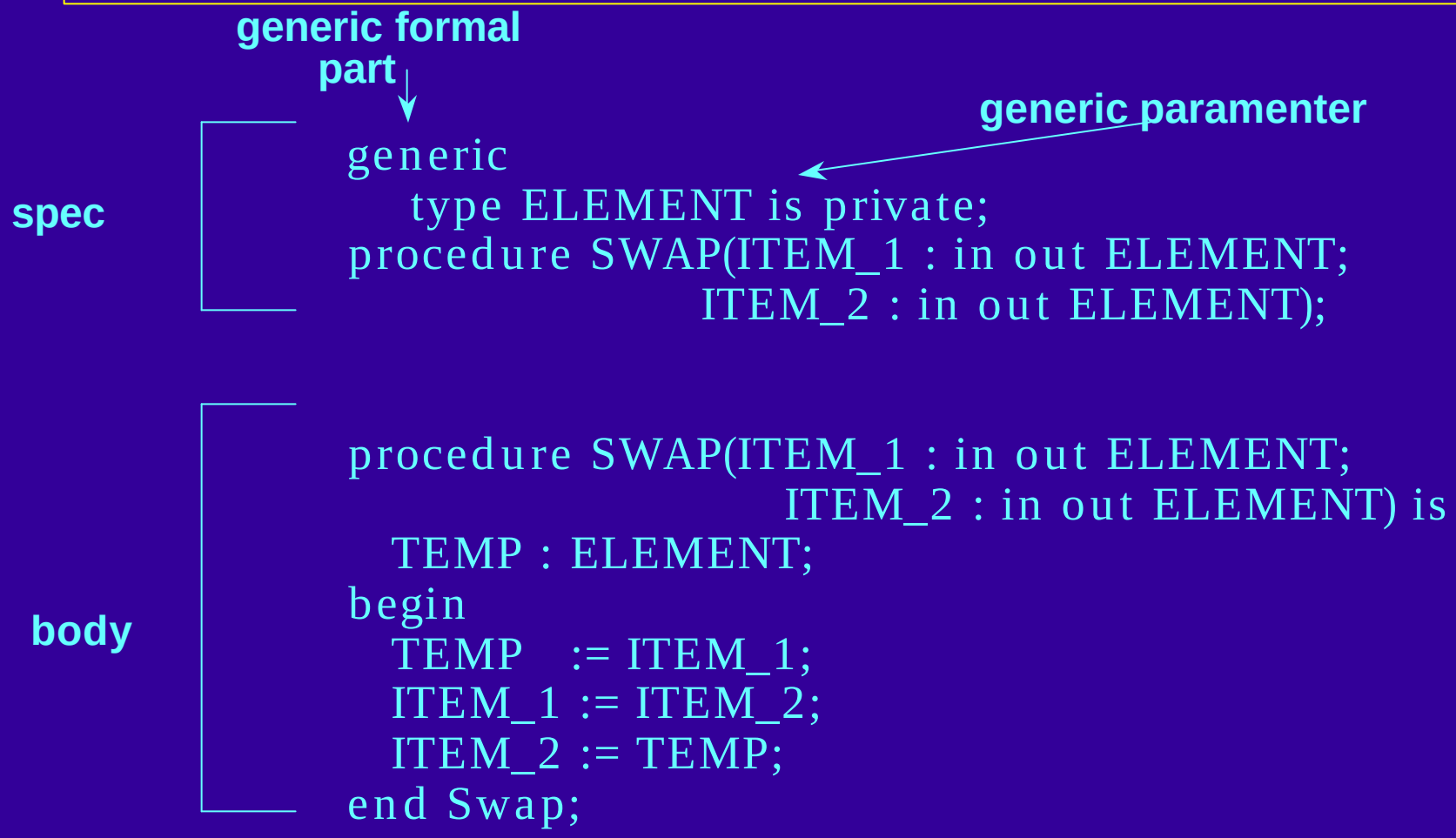


With a generic, we can pass a type and reuse one general routine or algorithim for a variety of data types.

Non-Generic Swap Routine

```
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;
```

Generic Swap Example



Using the Generic

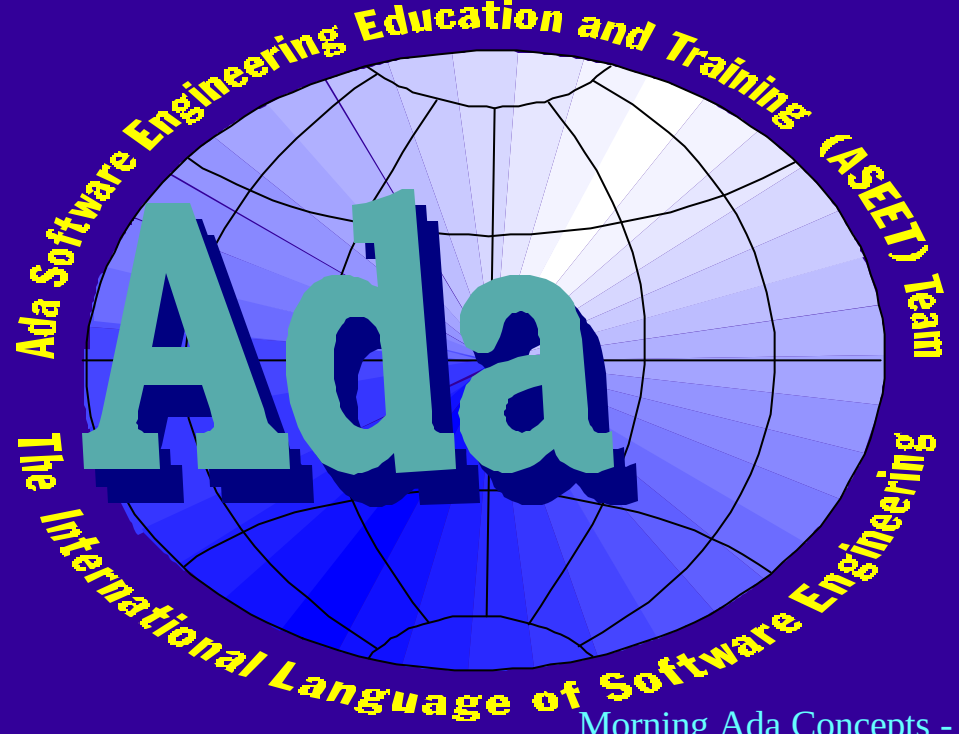
```
with Swap;
procedure Example is
    procedure Int_Swap is new Swap(Integer);
    procedure Chr_Swap is new Swap(Character);
    Num1, Num2 :  Integer;
    Char1, Char2 :  Character;
begin
    Num1 := 10;
    Num2 := 3;
    Int_Swap (Num1, Num2);

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

Summary

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

Introduction to Ada95

ANSI/ISO/IEC-8652:1995

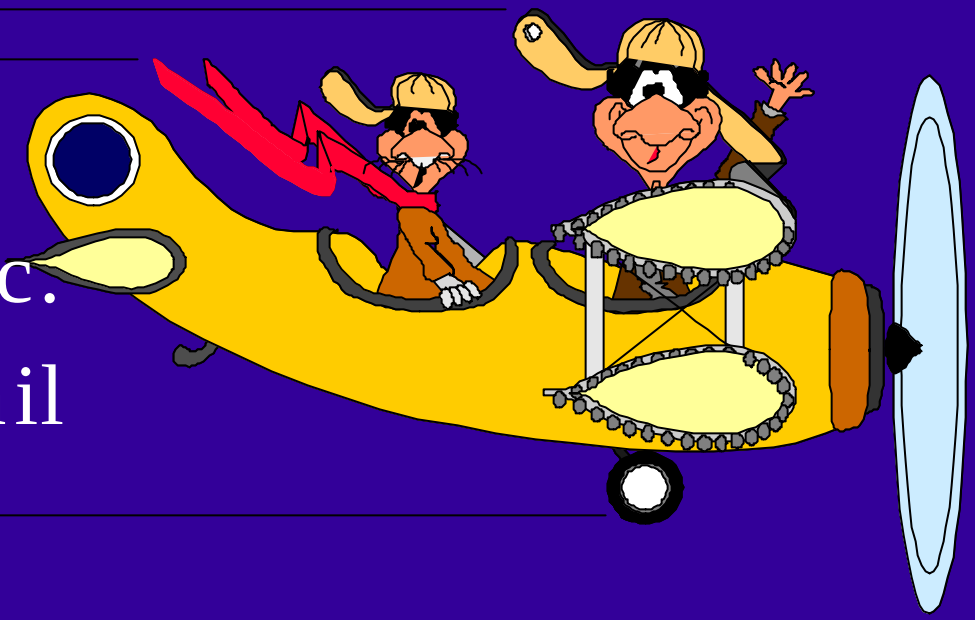


Speakers

Eugene Bingue, Ph.D.
Dr.Bingue@ix.netcom.com

David A. Cook, Ph.D.
USAF STSC/Shim Inc.
David.Cook@hill.af.mil

Leslie (Les) Dupaix
USAF STSC
Les.Dupaix@hill.af.mil



Introduction to Ada 95

- Why Ada 95?
- Overview of Ada
- Annexes

Why Ada 95?

A revision of ANSI/MIL-STD-1815A-1983 to reflect current essential requirements with minimum negative impact and maximum positive impact to the Ada community.

The Standard dropped the Mil-STD designation identification and now is ANSI/ISO/IEC-8652:1995.

Software Engineering for the 21st Century

The Main User Needs

- Program Libraries
- Object Oriented Programming
- Parallel & Real-Time Processing
- Interfacing

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

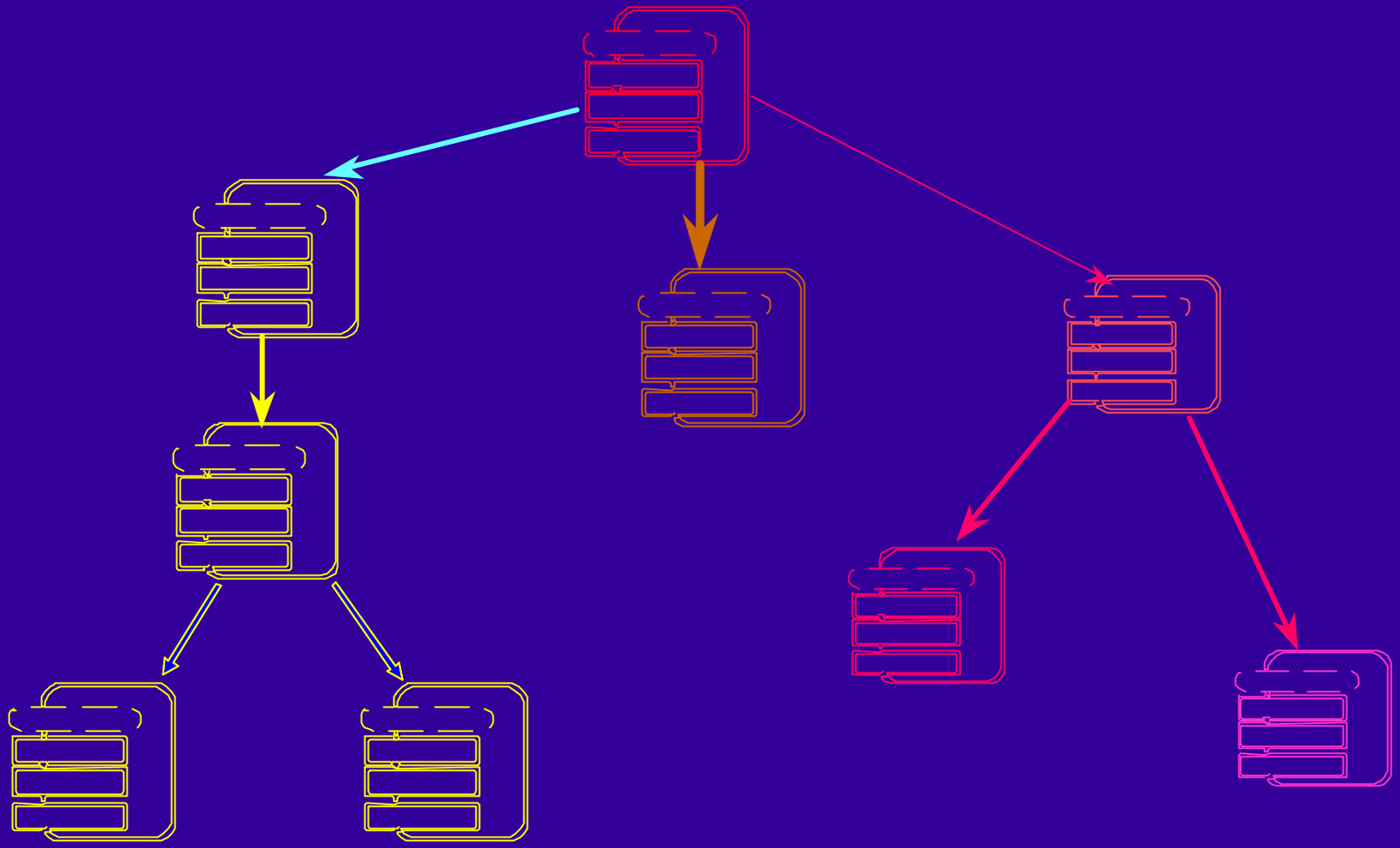
- loop
  - exit
  - for
  - while

OTHER

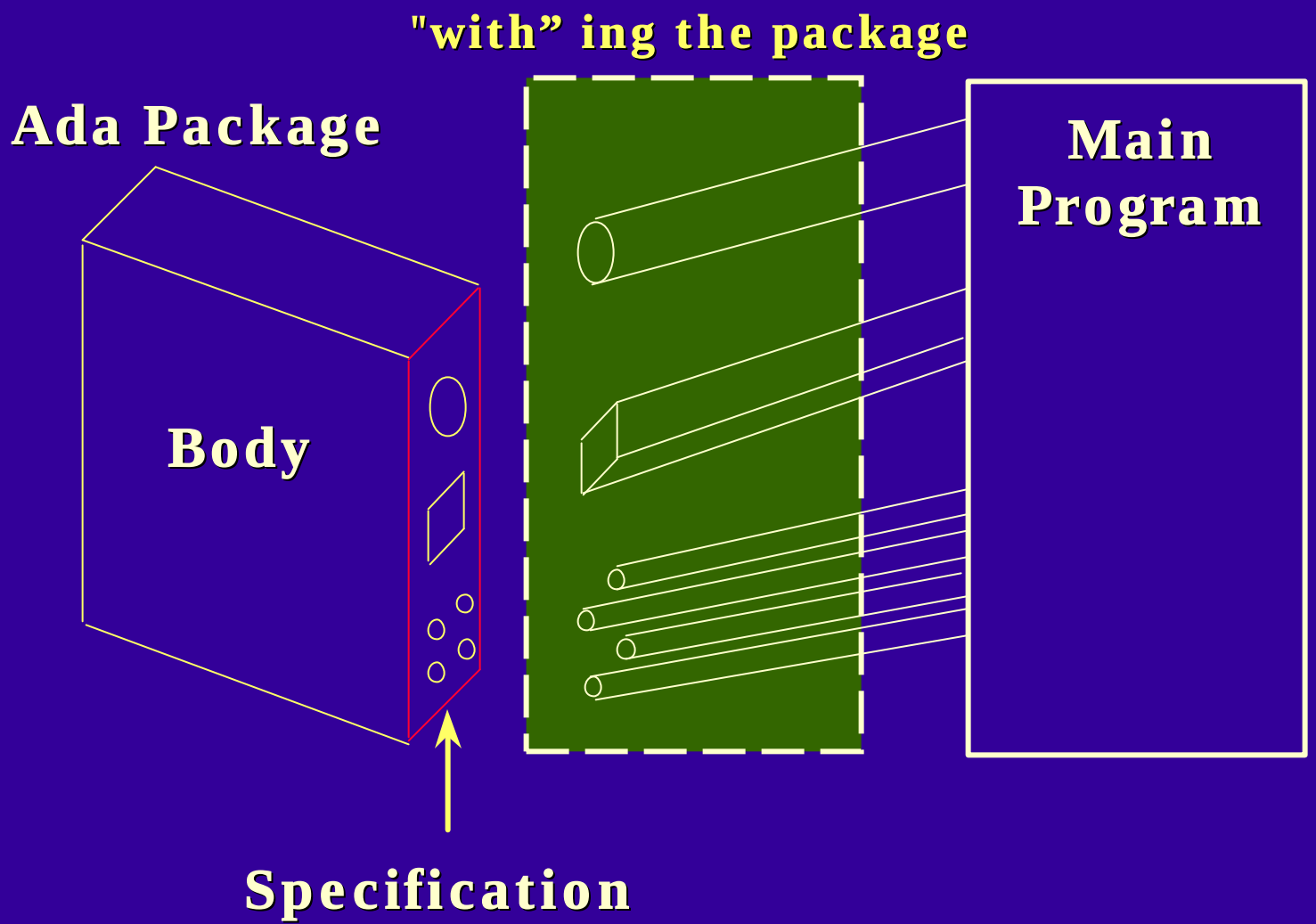
- raise
- goto

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

Hierarchical Libraries “Packages”



Specification and Bodies



Program Libraries

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

The flat nature of the Ada 83 library gave problems of visibility control. It prevented two library packages from sharing a full view of a private type. Resulting packages became large and monolithic maintenance nightmares.

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;
```

If you want to add additional features to the package (say, Polar notation), you want to do so in a manner that minimizes side effects.

Users who already *with* the package should not have to recompile, nor should they find additions to their "name space".

Hierarchical Libraries

Solution? Create a package *subordinate* to Complex_Numbers. It can use the resources of Complex_Numbers, but is a separate package in its own right. This is referred to as a *child* package.


```
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;
```

Rules for the *with* clause

- *with clause* - used to give visibility to a library unit
- This allows you to access components of Complex_Numbers using *dot* notation (formally known as *selected component notation*).
- The *with* clause gives visibility to anything found in the specification of the *with*ed unit

```
with Complex_Numbers;

procedure CALCULATE is
   My_Complex : Complex_Numbers.Complex;
begin
   My_Complex := Complex_Numbers. Cartesian_To_Complex(5.0, 2.0);
end;
```

Rules for the *use* clauses

- *use clause* - used to give direct visibility to a library unit
- Requires a *with* clause first
- This allows you to access components of Complex_Numbers without using *dot* notation

```
with Complex_Numbers;  use Complex_Numbers;

procedure CALCULATE is
        My_Complex : Complex;
begin
        My_Complex := Cartesian_To_Complex(5.0, 2.0);
end;
```

Don't Use the *Use*

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

Where Context Clauses Go

In the Specification

```
with Complex_Numbers;
package More_Math is.....
```

➔ **This implies that users of More_Math will also need Complex_Numbers**

In the Body

```
package More_Math is...
          ....

with Complex_Numbers;
package body More_Math is
          .....
```

➔ **This implies that the user of More_Math needs no additional resources, but that the implementation of More_Math (which is hidden from the user) uses Complex_Numbers internally**

Hierarchical Libraries

- If you *with* the child you have automatic visibility of the parent (but not direct visibility). In essence, a *with* clause for a child automatically includes a *with* for all ancestors.
- If you *with* the parent, you *DO NOT* have any visibility of any children, unless the children are *with*ed separately.
- The child has visibility of the parent.
- A child may *with* previously compiled siblings
- The child body has complete visibility to the parent body
- The parent body may *with* children
- The parent spec may not *with* any children

Hierarchical Libraries - private children

- A private child is typically used to add additional functionality to the parent.
- They prevent the parent from growing too large.
- Private children can only be seen by the *bodies* of their ancestor.
- Typically, they are *with*ed by the body of their parent.
- A private child is never visible outside of the *tree* rooted at the parent.
- In essence, the first *private child* down a long *chain* hides anything below it from outside view.

Private Children

```
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;
```

Child

```
private package Complex_Numbers.Hidden_Operations is
        --Types and procedures in this package can be used
        --in the body of Complex_Numbers.
end Complex_Numbers.Hidden_Operations;
```

Problems with Children

- ☹ They whine, stay on the phone too much, ... wait, wrong analogy!
- ☹ A child has automatic visibility of the complete specification of all of its “ancestors.
- ☹ This means that a child has full access to the private parts of the parent, grandparent, etc.
- ☹ If I desire access to the private parts of package Cant_Touch_Me, all I have to do is name my package Cant_Touch_Me.Got_Ya, and visibility is mine!!
- ☹ This is a configuration management issue. Simple tools can track which packages are children of other packages.

Friends don't Let Friends use *Use*

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

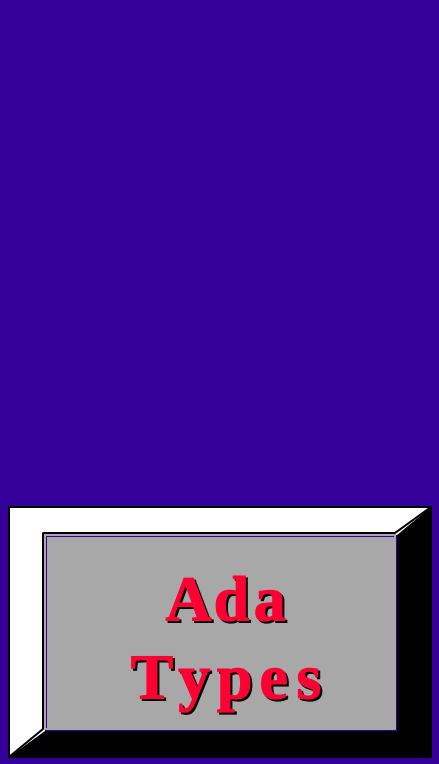
When Might it Help?

To achieve direct visibility of a *type*, not a *library unit.*

```
with Complex_Numbers;
procedure Test is
    use type Complex_Numbers.Complex;
   A,B,C :  Complex_Numbers.Complex;
                --NOTE:  Fully specified name required
 begin
   ...
   C := A * B;       -- infix operation allowed!!
```

The *new* use clause allows you to create direct visibility at a type rather than a package level. This allows infix operations on variables where the type is not directly visible.

Classes of 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

Predefined Types

```
Boolean
Integer
   Natural  (Subtype)
   Positive (Subtype)
   Mod      (Modulus) --   type Byte  is Mod 255;
   New                       -- an unsigned
                             byte

Float
Character                  -- ISO 8859-1 character set
Wide_Character     --      -- ISO10646 BMP character
   New                          set
String                     Bname : String  := "Bill “;
                                --String auto-set
                                to 5
Wide_String        --
   New                     Foo   : Wide_String := “Joe“;
```

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
                                         --parameterless 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

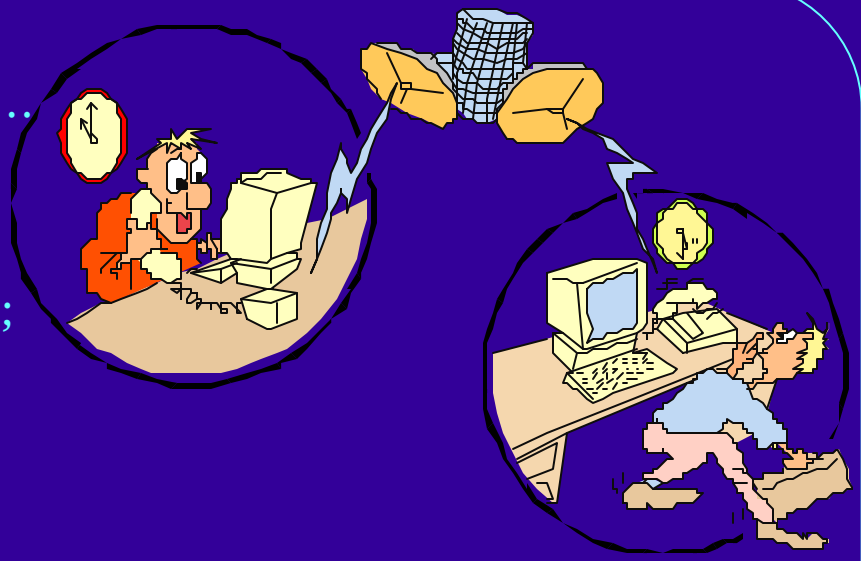
An access type can refer to a subprogram; an access-to-subprogram value can be created by the **'Access** attribute. A subprogram can be called using this pointer. This allows you to include a pointer to a routine inside of a record, or as a parameter. This is known as a ***callback***.

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

T : Trig_Function;

X, Theta : Float;

T := Sin'Access;

X := T(Theta);     -- implicit de-referencing.  SHOULD NOT BE USED!
                   -- This looks like normal function call.

X := T.all (Theta); -- explicit de-referencing.  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.

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 );
```



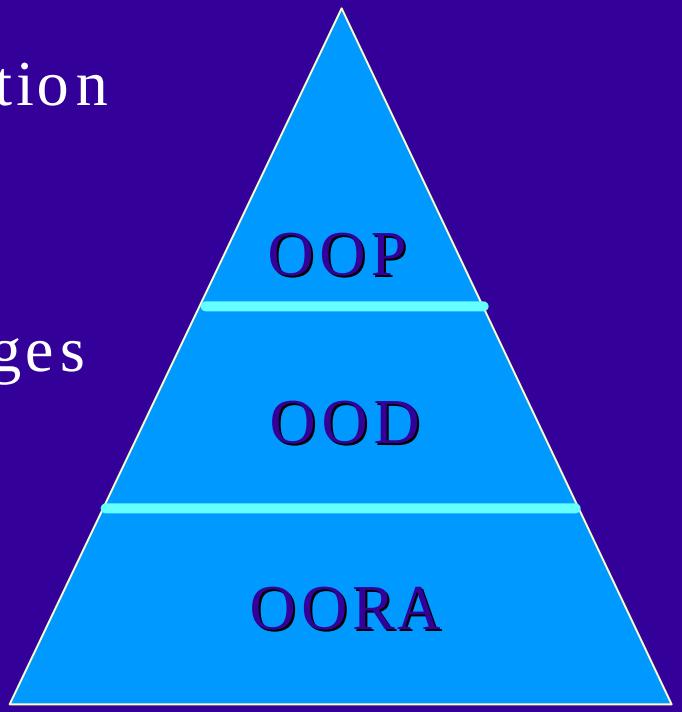
Object-Oriented Methods

OO is closely allied to reusability. It's main advantage is that systems can be separated into logical components, allowing better modeling of the problem world. OO creates better *abstractions*.

In addition, OO systems can be extended, rather than modified, to add functionality. This prevents disturbing existing software, eliminating the risk of introducing errors.

Ada Support for Phases of the OO Lifecycle

- OORA
  - Packaging
  - Abstraction & Encapsulation
  - Parallel Processing
- OOD
  - Packaging & Child Packages
  - Strong Typing
  - Enumeration Types
  - Parallel Processing
- OOP
  - Inheritance
  - Polymorphism (Dispatching)
  - Tasking



Tagged Type

```
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 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)) * 2.0 + ( 4.0 * C.Height);
end Perimeter;
```

Cuboid *inherits* Perimeter from Rectangle (technically, Perimeter is a *primitive* operation). 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.

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 does not have a body).
- An abstract subprogram MUST be overridden for EACH subclass

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 it's 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;
```

Building Inheritance Chains

- All record that derive from a tagged record are implicitly tagged.
- Each inherited record inherits all fields and operations from its parent, creating an inheritance *chain*.
- If you use an operation on a inherited type that is not explicitly written for that type, then the *chain* is searched towards the root. The first instance of the operation will apply.
- You can even add abstract records at a child level, allowing you to selectively

Abstract types

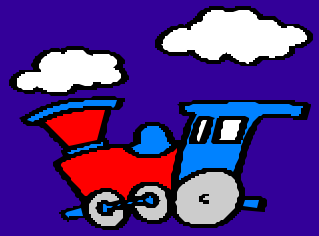
```
type Planes is abstract new Vehicle with
        record
                Wingspan : Some_Type;
        end Planes;

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

You cannot declare a variable of type Planes (it is abstract), so you must derive from it. However, when you derive a new type from Planes, you must also override the function Runway_Needed_To_Land.

Polymorphism

- From the Greek *poly*, many, and *morphe*, form. Polymorphism is another property of Object Oriented languages.
- Class-wide types are said to be polymorphic because a class-wide variable can hold objects belonging to any type within the class.

Class Wide Programming type T'Class



Vehicle Vehicle'Class

Trains **Automobiles**

Planes Planes'Class

Jet

Propeller

Helicopters

- With each tagged type there is an associated type 'Class.
- The values of this 'Class type include all derived types.
- Any derived type may be converted to the type 'Class.

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)

   Start (Jet(Item));     -- static call to the Start for Jet.
                          -- this call will fail at run time if Item is not
                          -- a member of the Jet hierarchy

   ...
end Move_All;
```

Class Wide Programming (Dispatching using pointers)

```
-- Vehicles held as a heterogeneous list using an access type.
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 de-referencing of pointer
end Move_All;
```

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. Every time 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;
```

```
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.
```

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 and then read
  - in out - Parameter may be both read and updated

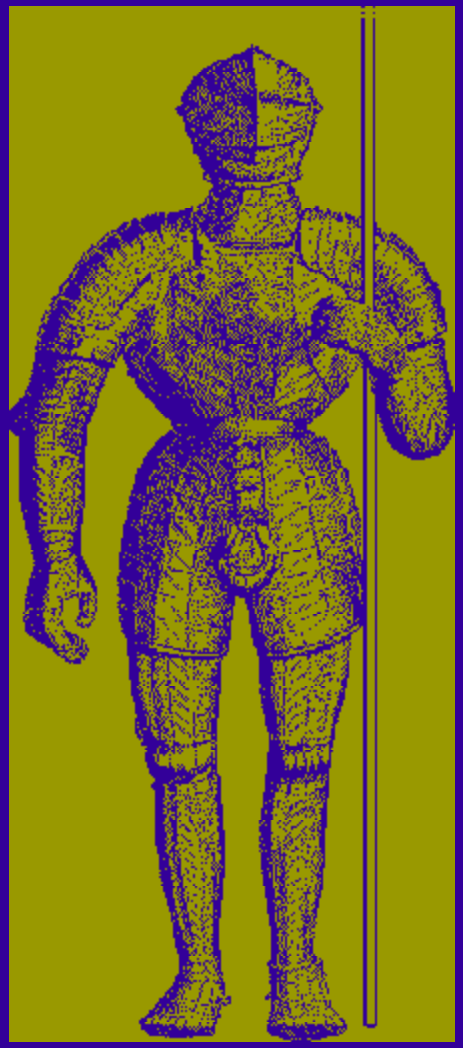
must be a variable

- Functions may only have *in* parameters

Parallel Processing (Tasking)

- The Ada parallel processing model is a useful model for the abstract description of many parallel processing problems. In addition, a more static monitor-like approach is available for shared data-access applications.
- Ada provides support for single and multiple processor parallel processing, and also includes support for time-critical real-time and distributed applications.

Protected Types



Protected types provide a low-level, lightweight synchronization mechanism whose key features are:

- Protected types are used to control access to data shared among multiple processes.
- Operations of the protected type synchronize access to the data.
- Protected types have three kinds of operations: protected functions, protected procedures, and entries.

Protected Units & Protected Objects

- Protected procedures provide mutually exclusive read-write access to the data of a protected object
- Protected functions provide concurrent read-only access to the data.
- Protected entries also provide exclusive read-write access to the data.
- Protected entries have a specified barrier (a Boolean expression). This barrier must be true prior to the entry call allowing access to the data.

Protected Types

```
package Mailbox_Pkg is
    type Parcels_Count is range 0 .. Mbox_Size;
    type Parcels_Index is range 1 .. Mbox_Size;
    type Parcels_Array is array ( Parcel_Index ) of Parcels
    protected type Mailbox is
                -- put a data element into the buffer
        entry Send (Item : Parcels);
                -- retrieve a data element from the buffer
        entry Receive (Item : out Parcels);
        procedure Clear;
        function Number_In_Box return Integer;
    private
        Count           : Parcels_count := 0;
        Out_Index       : Parcels_Index := 1;
        In_Index        : Parcels_Index := 1;
        Data            : Parcels_Array ;
    end Mailbox;
end Mailbox_Pkg;
```

Protected Types Example

```
package body Mailbox_Pkg is

  protected body  Mailbox is

     entry Send ( Item : Parcels) when Count < Mbox_Size is
                                          -- block until room
     begin
        Data ( In_Index ) := Item;
        In_Index := In_Index mod Mbox_size + 1;
        Count := Count + 1;
     end Send;

     entry Receive ( Item : out Parcels ) when Count > 0 is
                                           -- block until non-empty
     begin
        Item := Data( Out_Index );
        Out_Index := Out_Index mod Mbox_Size + 1;
        Count := Count -1;
     end Receive;
```

Protected Types
Example (cont.)

```
        procedure Clear is              --only one user in Clear at a time
        begin
            Count := 0;
            Out_Index := 1;
            In_Index := 1;
        end Clear;

        function Number_In_Box return Integer is
                                        -- many users can check # in Box
        begin
            return Count;
        end Number_In_Box;

    end Mailbox;

end Mailbox_Pkg;
```

Asynchronous Transfer of Control

```
select
      triggering_alternative;
then abort
      abortable_part;
end select;
```

- The abortable part is executed if the triggering alternative is not ready.
- If the triggering alternative becomes ready prior to the completion of the abortable part, then the abortable part is aborted.
- If the abortable part completes, then the triggering alternative is cancelled.

Asynchronous Transfer of Control (Waiting for an Event)

```
loop
   select
      Terminal.Wait_For_Interrupt;
      Ada.Text_IO.Put_Line("Interrupte
        d");
   then abort
      Ada.Text_IO.Put_Line("Enter
        Command   --> ");
      Ada.Text_IO.Get_Line(Command,
        Last);
      Parse_and_Process (Command
        (1..Last) )
```

Asynchronous Transfer of Control (Creating a *Timeout*)

```
select
        delay 5.0;
        Ada.Text_IO.Put_Line("Calculation does not converge");
        Some_Default_Action;
    then abort
        Horribly_Complicated_Recursive_Function(X,Y);
    end select;
    -- After 5 seconds (plus a little), I will reach here
```

Requeue Statement

```
requeue Entry_Name [with abort];
```

- The *requeue* allows a call to an entry to be placed back in the queue for later processing.
- Without the *with abort* option, the requeued entry is protected against cancellation.

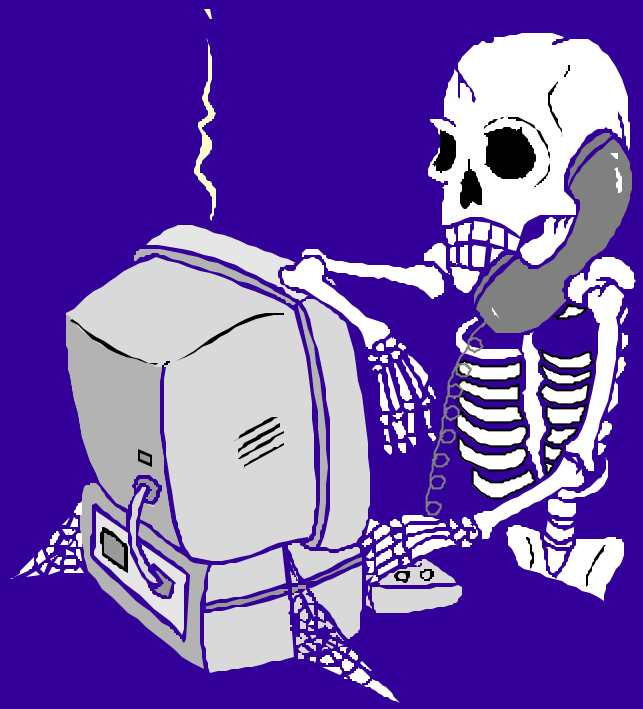
Delay and Until Statements

```
delay Next_Time - Calendar.Now;
```

suspended for at least the duration specified

```
delay until Next_time;
```

specifies an absolute time rather than a time interval



The *until* *does not* provide a guaranteed delay interval, but it does prevent inaccuracies due to swapping out between the “delay interval calculation” and the delay statement

EXCEPTIONS

Hadnling all posible Erors!

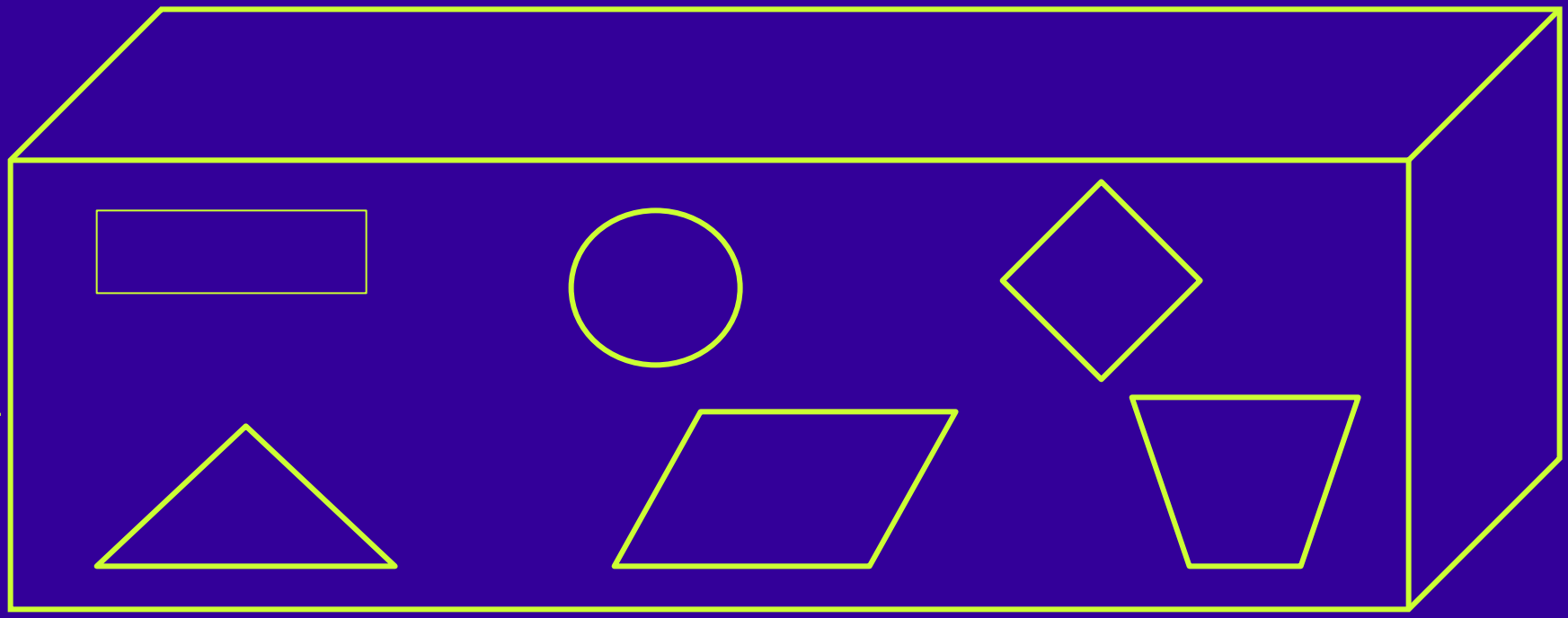
Exceptions and exception handling in Ada 95 are still exceptional!!

Note : There are four errors in this slide.

Exceptions

- Exceptions are a mechanism to allow the programmer to identify and handle errors within the program without calling system error routines.
- There are two kinds of exceptions:
  - *predefined exceptions*
    - Standard (*Constraint, Program, Storage, Tasking*)
    - IO (*Status, Mode, Name, Data, Layout, End, Device, Use)*
  - user-defined exceptions
    - Allows the designer to define and *raise* non-standard exceptions
- *Exception handlers* handle the exception.
  - You can also *name* and *save* an exception externally.
  - An exception can be *re-raised* and it can be

GENERICS



Generics Define a Template Or Mold for Programs Units

Generics

- Generics allow you to create a template for packages, functions, and procedures.
- Generic templates can be customized via *runtime elaboration*.
- During elaboration, generic parameters are used to create an *instantiation* of the generic template.
- Generic parameters include
  - types
  - variables and constants
  - functions and procedures
  - another generic instantiation
  - packages

Unchecked Conversion and Unchecked Access

- Unchecked_Conversion
  - Converts any type to any other type.
  - Strictly a *bit by bit* conversion.
  - Constraint and Range checking left up to the program.
  - This attribute, if misused, can lead to corrupted data structures.
- X'Valid
  - Checks to see if the object has a *safe* value.
- X'Unchecked_Access
  - Overrides type checking for access values.
  - Program is responsible for removing local objects from global data structures prior to exiting the objects' scope.

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. Predefined Language Environment

Annex A contains packages that support

- Predefined Identifiers
- Character Handling
- String Handling
- Numerical Functions
  - Basic math functions
  - General Trig/Log functions
- Random Number Generation
  - Discrete
  - Continuous

INPUT/OUTPUT

- Ada supports many different types of predefined IO.
- Each type has its' own package with supporting functions and procedures:
  - Ada.Sequential_IO
  - Ada.Direct_IO
  - Ada.Wide_Text_IO
  - Ada.Streams.Stream_IO
  - Ada.Text_IO --this is where *Text_IO is now*

Other “Bells and Whistles”

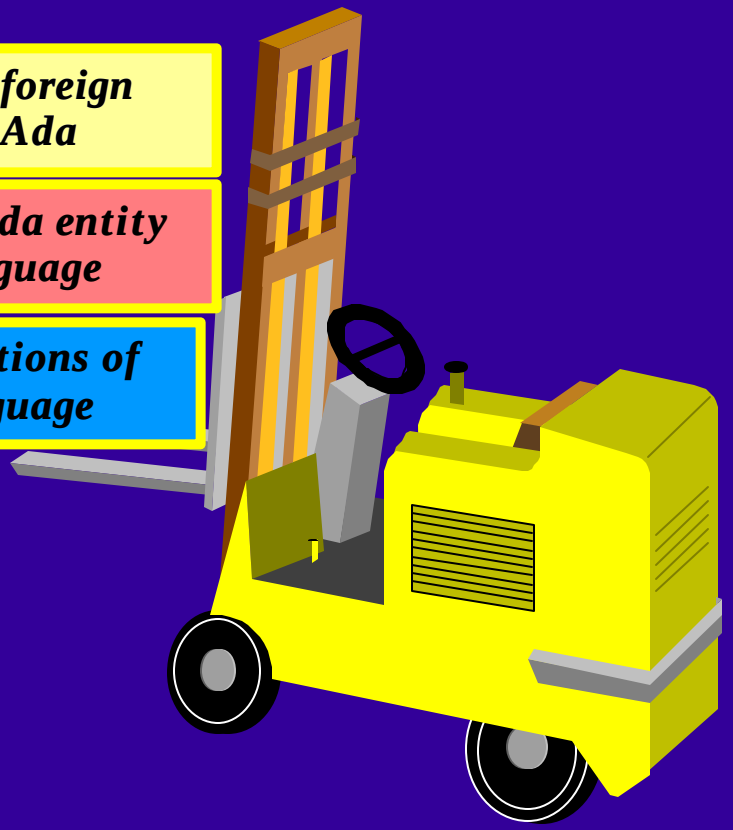
- One character look-ahead in files (without moving file pointer)
- Get immediate from keyboard (no carriage return required)
- Procedure to *flush* file buffer without closing then re-opening file
- Stream IO (mixed text and binary)
- Ability to read command line from within program

Annex B
Interfacing to Other Languages

pragma Import ← *used to import a foreign language into Ada*

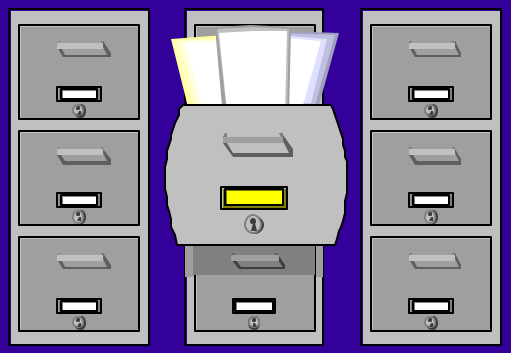
pragma Export ← *used to export an Ada entity to a foreign language*

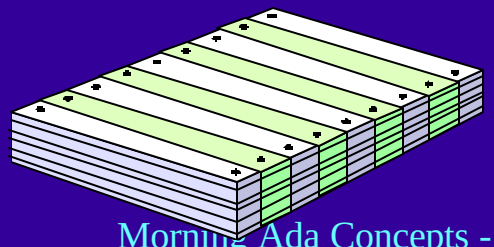
pragma Convention ← *use the conventions of another language*



Standard Interfaces - Required Packages

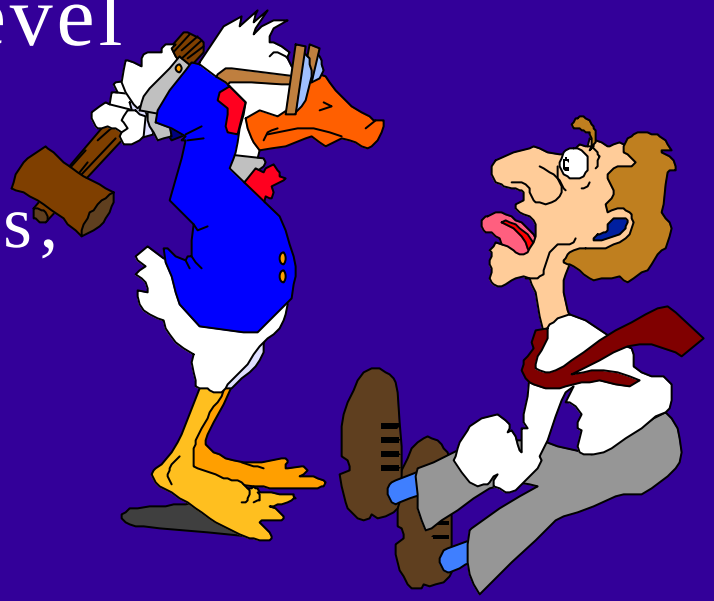
- package Interface.C
  - Has two required child packages
    - Interface.C.Strings
    - Interface.C.Pointers
- package Interface.Cobol
- package Interface.Fortran





Annex C. Systems Programming

- Covers a number of low-level features such as
  - in-line machine instructions,
  - interrupt handling,
  - shared variable access
  - task identification.
  - *Atomic* pragma (indivisible read/writes)
  - *Volatile* pragma (bypasses cache memory)
- This annex is a requirement for Annex D, Real-Time Systems Annex.



Annex D. Real-time Systems

- Includes pragmas that allow you to tailor
  - scheduling of parallel processes
  - priorities of parallel processes
  - queueing protocols for entry calls
  - ceiling-locking protocols
- Must include documentation specifying
  - time it takes to actually abort a task on both single and multi-processor systems
  - time it takes to process an asynchronous select
- Includes a Monotonic time package
- Includes low-level asynchronous and synchronous task control options

Annex E. Distributed Systems

- Includes features that give you ability to
  - communicate between *partitions* running on different *processing* and/or *storage* nodes.
  - categorize library units as to how they are used (determines if/when it can be distributed).
  - set up a remote library that is used for remote procedure calls (RPCS), using both *static* binding and *dynamic* binding of remote procedures
  - make an *asynchronous* procedure call (which returns without waiting for

Annex F. Information Systems

This Annex provides a set of of facilities relevant to Information Systems programming. These fall into several categories:

- The package Decimal which declares a set of constants defining the implementation's capacity for decimal types, and a generic package for decimal division.
- The child package Text_IO.Pictures, which supports formatted and localized output of decimal data, based on *picture string*.

Annex G. Numerics

- The Numerics Annex specifies:
  - features for complex arithmetic, including complex I/O
  - a *strict* mode in which the predefined arithmetic operations of floating point and fixed point types have to provide guaranteed accuracy or conform to other numeric performance requirements
  - a *relaxed* mode in which no accuracy or other numeric performance requirements need be satisfied (as for implementations not conforming to the Numerics Annex)

Annex H. Safety and Security

- This Annex address requirements for systems that are safety critical or have security constraints. It provides facilities and specifies documentation requirements that relate to several needs:
  - Predicting program execution
  - Reviewing of object code
  - Restricting language constructs whose usage might interfere with program reliability

Pragma Reviewable;

- Directs the compiler to generate object code that can be independently validated.
- The following information must be produced
  - Where compiler-generated run-time checks remain
  - Identification of any construct that is certain to fail
  - Where run-time support routines are implicitly invoked
  - For each scalar, either “Initialized” or “Possibly uninitialized”
  - An object code listing with machine instructions, offsets, and source code correspondence
  - Identification of each construct with possible erroneous execution
  - Order of library elaboration

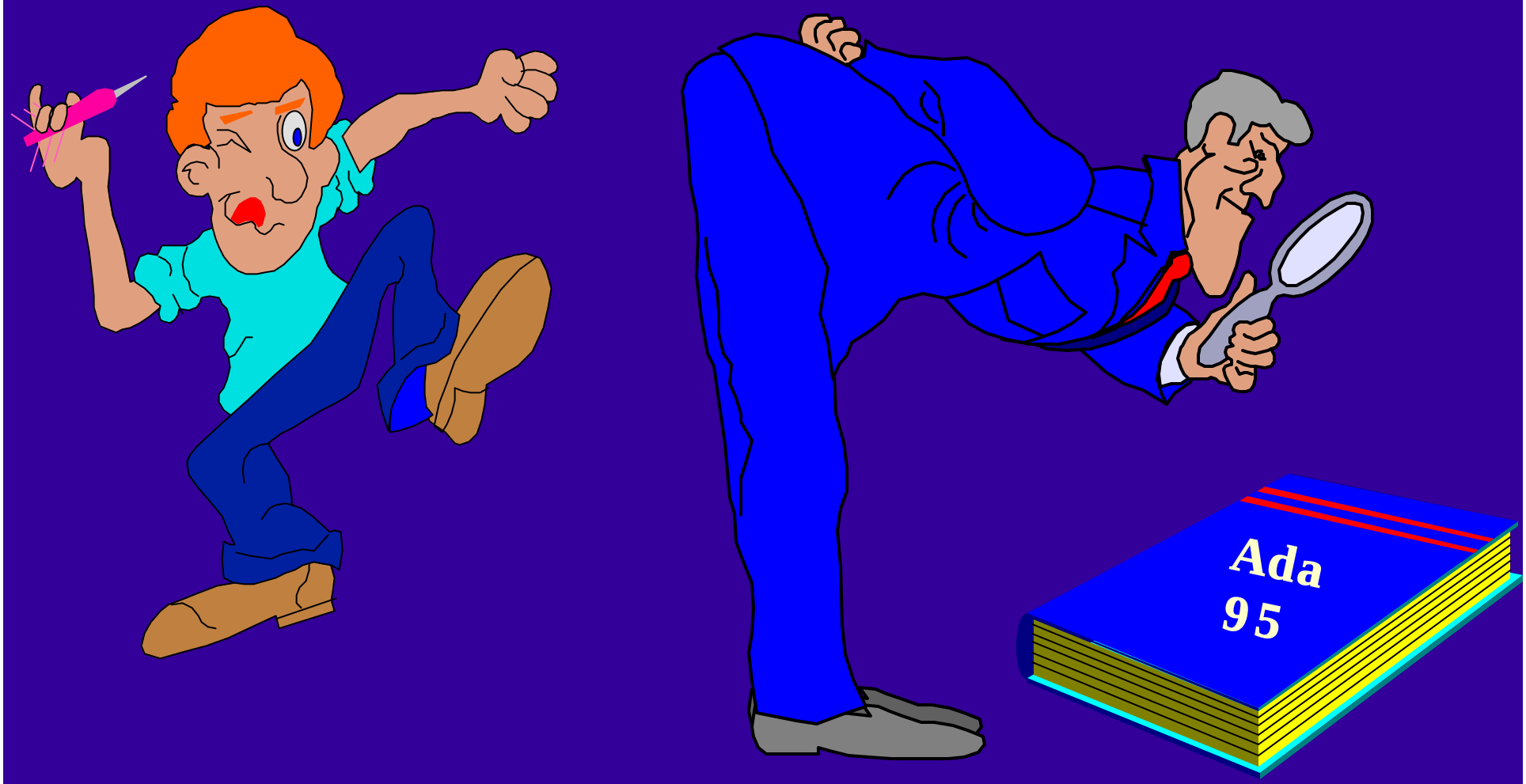
Annex J. Obsolescent Features

- This section contains descriptions of features of the language that worked under Ada 83, but are no longer needed and not recommended under Ada 95.
- Most good programmers will not find any of the “obsolescent features” a problem. However, there are a few changes to Ada 95 that would require a lot of “nit-picking” changes. There are a few predefined renaming clauses to prevent you from having to edit all of your old programs. However, your new programs should use the correct methods.

Example →

```
with Ada.Text_IO;
package Text_IO renames Ada.Text_IO;
```

Questions?



The End!