An Introduction to Ada 95 for Programmers

Dr. David A. Cook
DCook@AEgistg.com

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

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: 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)
- explicit 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.

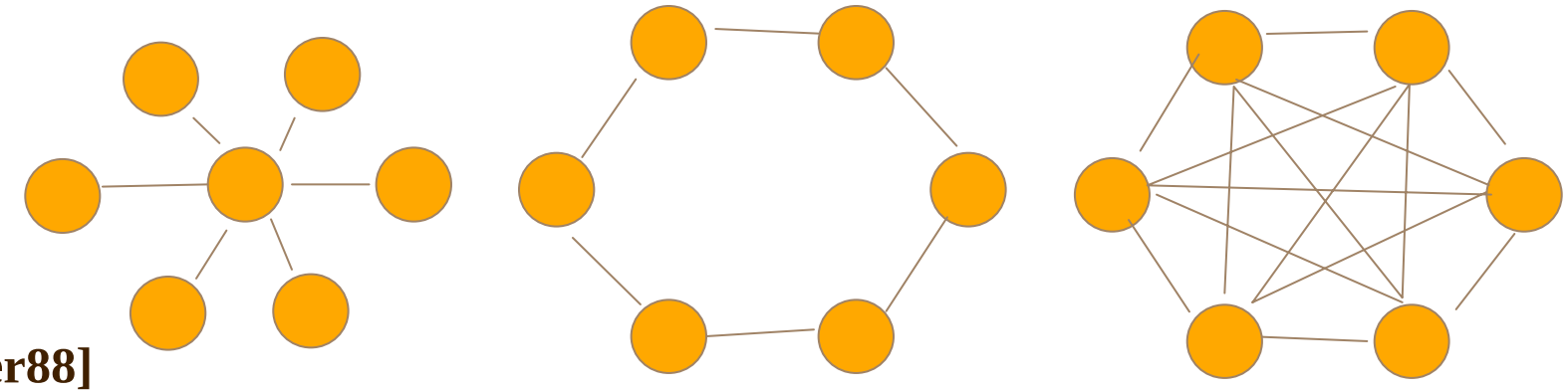
> Modules must correspond to syntactic units in the language used

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.



[Meyer88]

min: (n-1) one boss

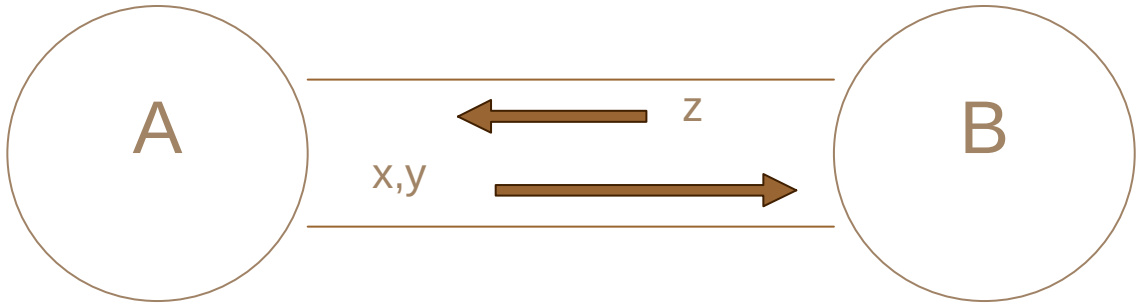
robust

max: n(n-1)/2

Every module should communicate with as few others as possible

Small Interface (Loss Coupling)

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

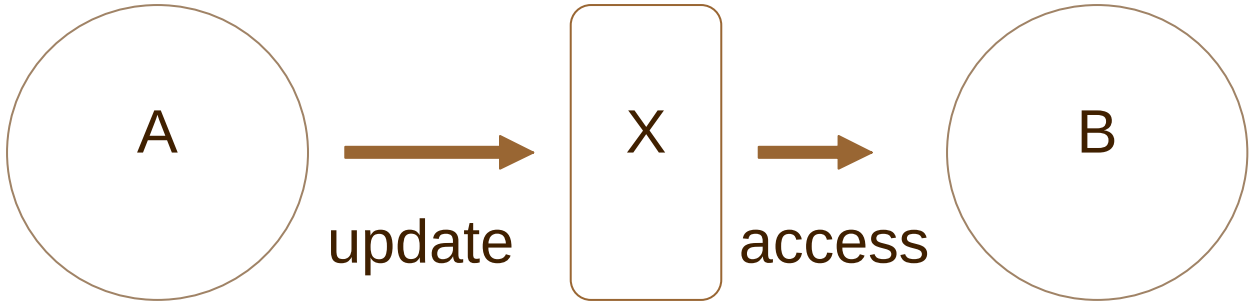


[Meyer88]

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

Explicit Interfaces

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

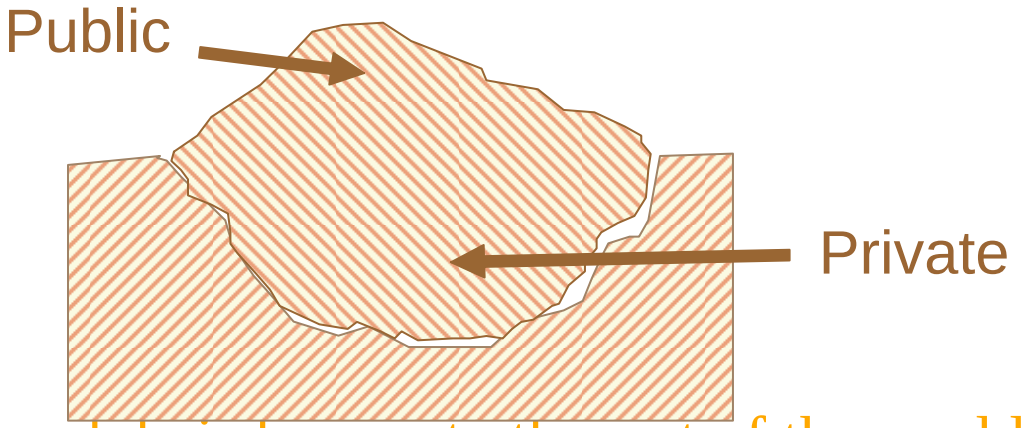


Data Sharing

[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.

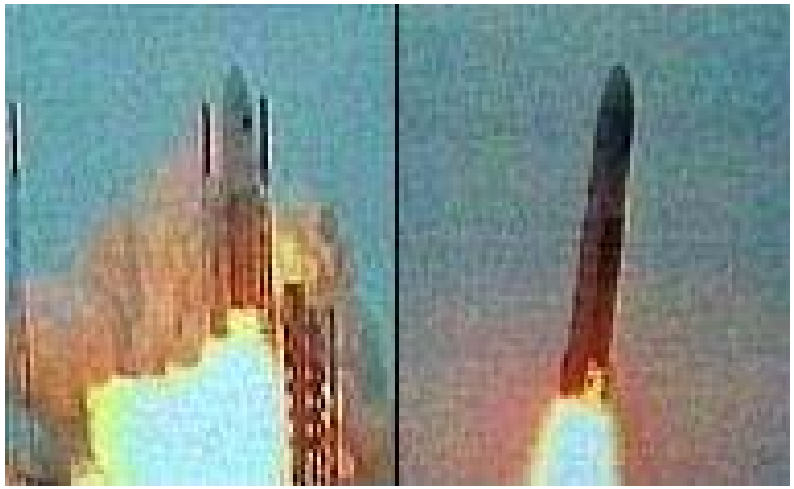
[Meyer88]

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



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

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

- What lessons might you draw so far?

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.***

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

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

Ada 95 Reference Manual (LRM)

1. General
2. Lexical Elements
3. Declarations and Types
4. Names Expressions
5. Statements
6. Subprograms
7. Packages
8. Visibility Rules
9. Task and Synchronization
10. Program Structure and Compilation issues
11. Exceptions
12. Generic Units
13. Representation issues

ANNEXES

A. Predefined Language Environment
B. Interface to Other Languages
C. Systems Programming
D. Real-Time Systems
E. Distributed Systems
F. Information Systems
G. Numerics
H. Safety and Security
J. Obsolescent Features
K. Language-Define Attributes
L. Language-Defined Pragmas
M. Implementation-Defined Characteristics
N. Glossary
P. Syntax Summary

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.

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

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''

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”

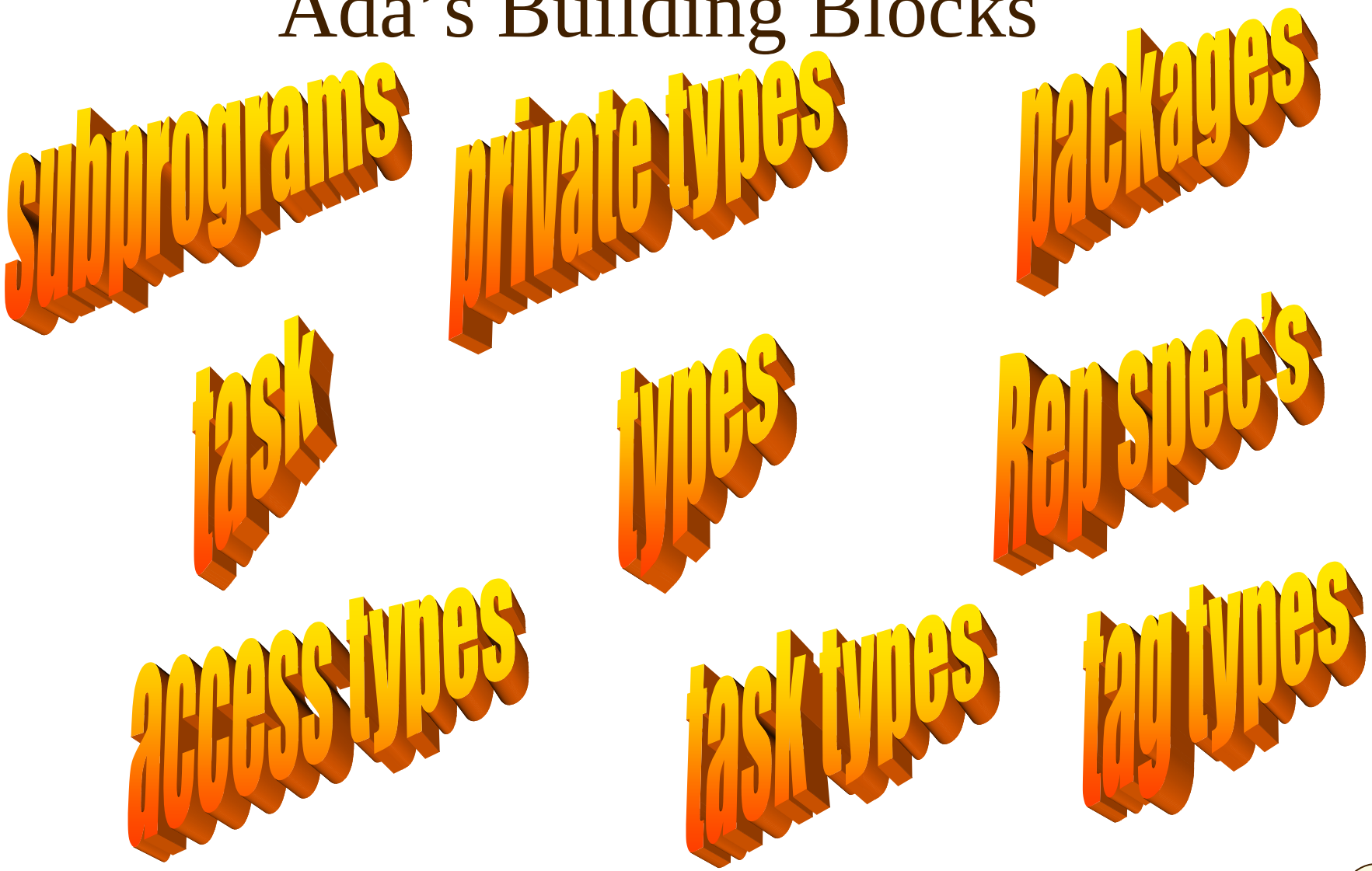
Basics of the language

- 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

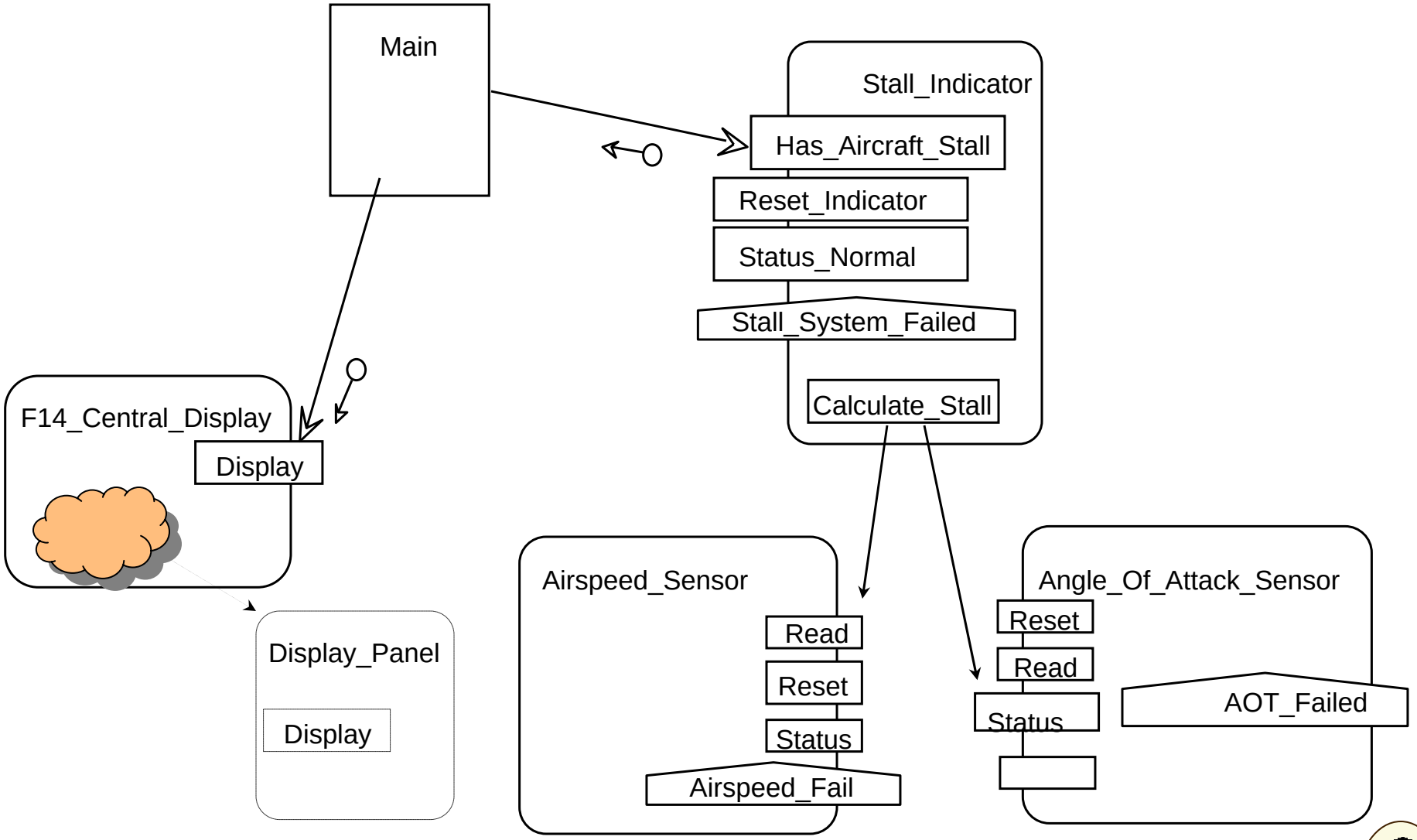
"The tools we use have a profound (and devious) influence on our thinking habits, and therefore, on our thinking abilities."

--E.W. Dijkstra

Ada's Building Blocks



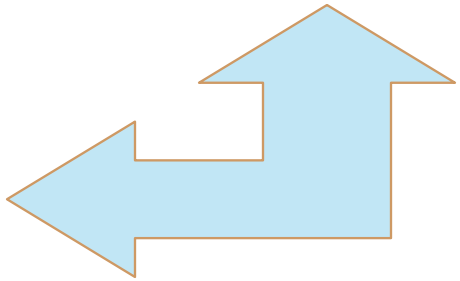
Anatomy of a Ada Program

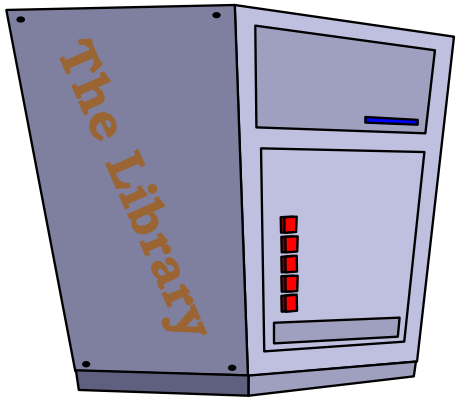


The Library

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

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

added in Ada 95

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
```

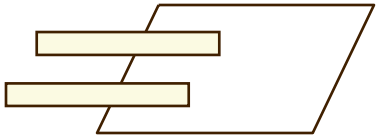
Program Units

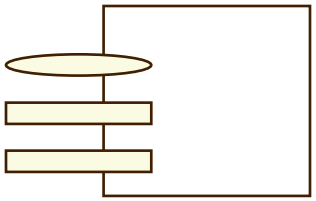
Subprograms
- -- Functions and Procedures
- -- Main Program
- -- Abstract Operations

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

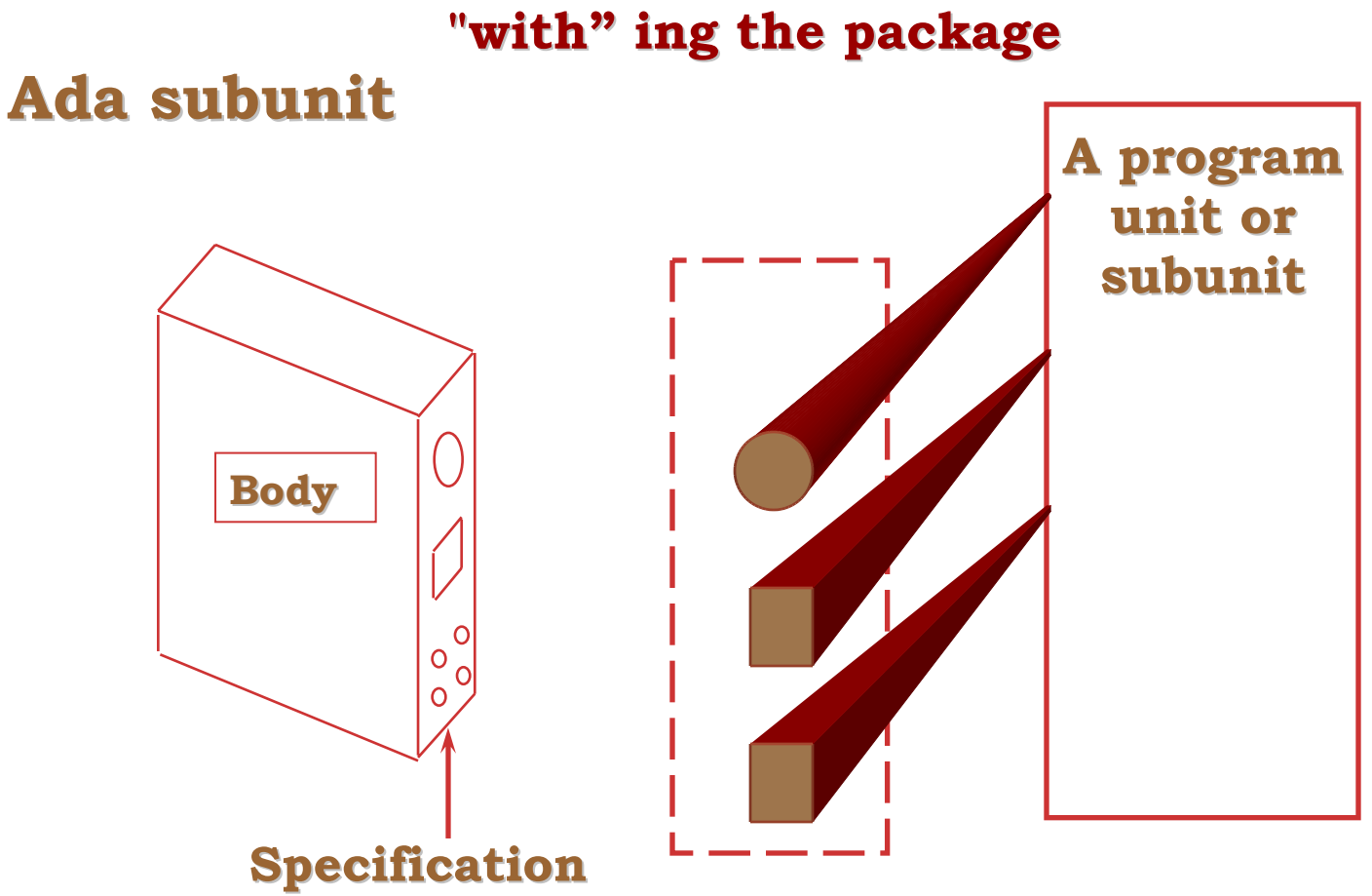
Packages
- -- Encapsulation
- -- Information Hiding
- -- Abstract Data Types

Generics
- -- Templates

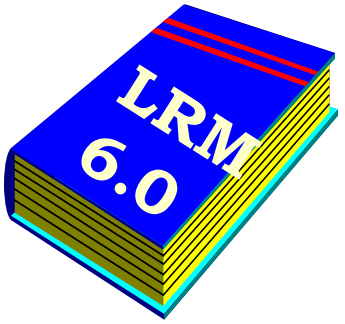




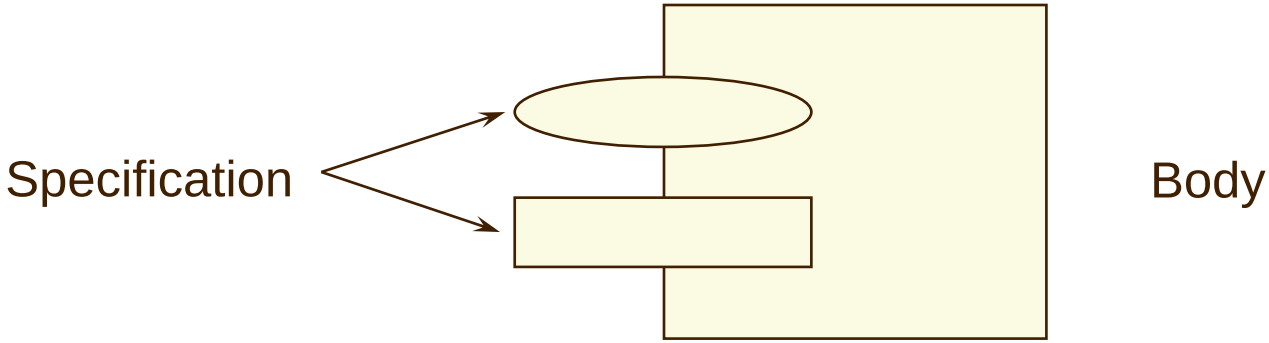
Specification and Bodies



Subprograms



Subprogram Units



"What" the program unit does

← **ABSTRACTION** →

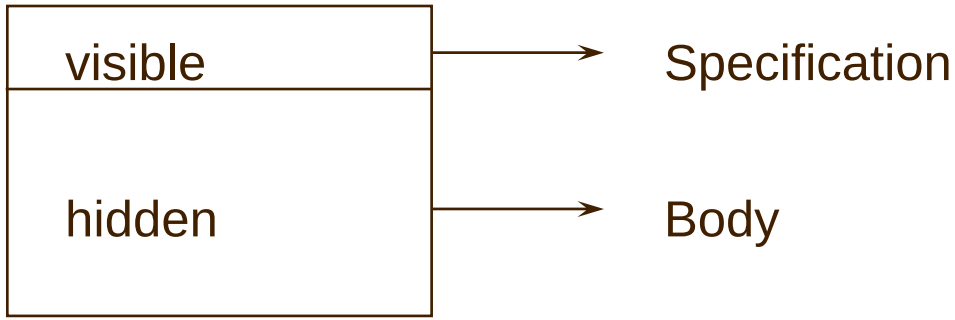
"How" the program unit does what it does

All the user of the program unit needs to know

← **INFORMATION HIDING** →

The details of implementation are inaccessible to the user

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);
```


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! It serves to
  - Inform the compiler about an available unit
  - Explain the interface to a developer who wants to use it

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. For some compilation units – it a body occurs without a specification, the specification is "assumed" to exist (as in the example above).

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, compilation error
    LOCAL_CHARACTER := SECOND;  -- okay
    SECOND := LOCAL_CHARACTER;  -- okay
    LOCAL_CHARACTER := THIRD;  -- illegal in Ada 93, OK in 95
    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) := others => ' ';
   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, can't switch from named to positional
```

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.

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);
```


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;
```

Function

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   NOTE that this is bad practice
--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.

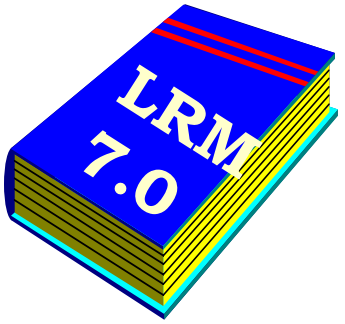
Summary (cont.)

- ✓ 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.)

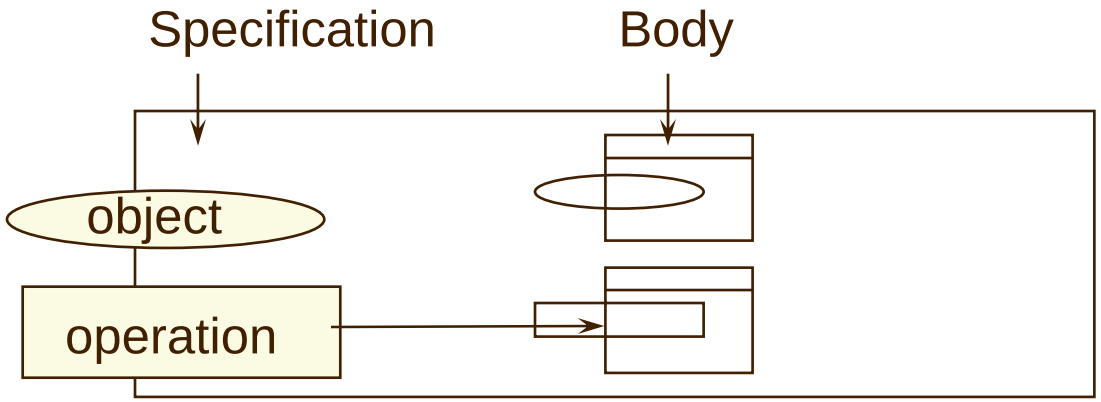
Exercise

- 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"

Packages



Ada Packages



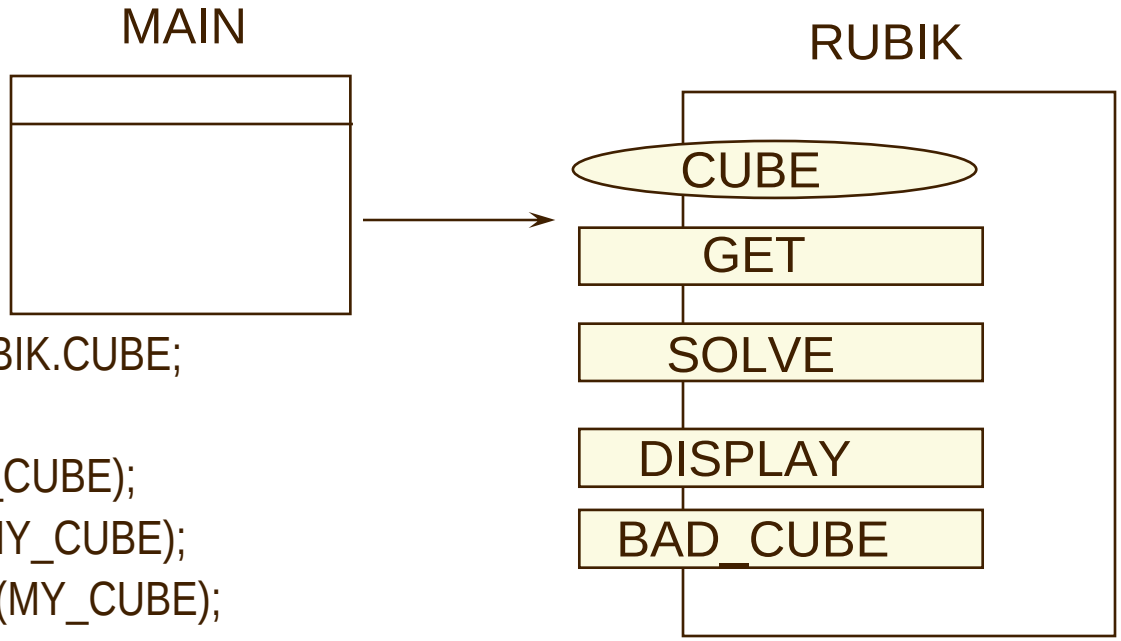
* The PACKAGE is the primary means of "extending" the Ada language.
* The PACKAGE hides information in the body thereby enforcing the abstraction represented by the specification.
* Operations (subprograms, functions, etc) whose specification appear in the package specification must have their body appear 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;
```

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; --from previous slide
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 linked or executed.

Visibility

A PACKAGE CAN BE MADE AVAILABLE IN TWO DISTINCT WAYS

-- It can be textually nested (seldom occurs)

-- It can be accessed from a library (most common usage)

```
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

Packages As Nested 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*

Bad Use Clause Example

```
With Ada.Text_io;
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
   Ada.Text_IO.put(Color'Image(Red));        --ERROR  (Why?)
   Ada.Text_IO.put(Color'Image(Color'(Red)));
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.