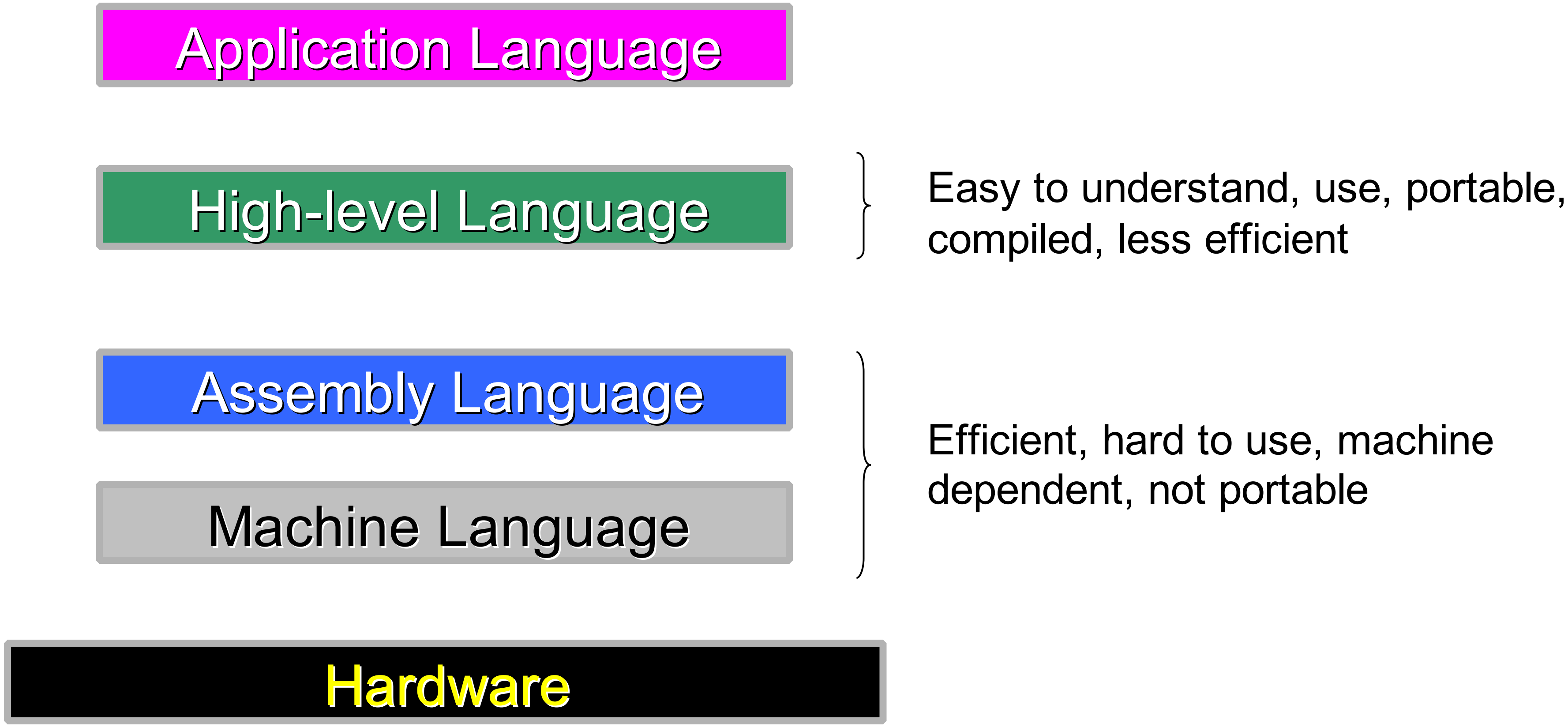
Ada 编程语言

2007/3

Programming Language



Ada 95

- Ada 95
  - Strong typing / run-time checking / parallel processing / exception handling / generics
  - Originally targeted for embedded and real-time systems
  - Now include aerospace and safety-critical systems

Use of Ada Around the World

- The control software of nearly every new commercial aircraft model, including the Boeing 777, the Airbus 340, and many regional airlines
- Nearly every country’s air traffic control system
- A number of communications and navigational satellites and ground-based equipment

Resources

- GNAT 3.15p (Free Software Foundation)
- AdaGide (US Air Force Academy)
- http://www.adahome.com/
- http://www.adapower.com/
- http://www.adaic.com/
- http://gcc.gnu.org/onlinedocs/gnat_rm/
- http://www.adahome.com/rm95/rm9x-toc.html

Hello World

hello.adb

```
WITH Ada.Text_IO;
PROCEDURE Hello IS
------------------------------------------------------------------------
--| A very simple program; it just displays a greeting.
--| Author: Michael Feldman, The George Washington University
--| Last Modified: June 1998
------------------------------------------------------------------------
BEGIN -- Hello
Ada.Text_IO.Put(Item => "Hello there. ");
Ada.Text_IO.Put(Item => "We hope you enjoy studying Ada!");
Ada.Text_IO.New_Line;
END Hello;
```

Common Programming Errors

- Compilation errors
- Run-time errors
- Logic or algorithmic errors

Compilation Errors

- Syntax errors
  - Fatal error that has to be fixed before code can be compiled
- Semantic errors
  - Inconsistency in the use of values, variables, packages, …

Run–time Errors

- Detected during execution of a program
- Called **exception** in Ada
- In Ada we have a way of predicting the occurrence of exceptions and prevent the computer from halting
  - Exception handling

Logic / Algorithm Errors

- Developing an incorrect algorithm for solving a problem
- Incorrect translation of a correct algorithm

The computer does only what you tell it to do,
not what you meant to tell it to do… (GIGO)

Comments, headers, and programming style

- Good programming style:

Communication

- Good style leads to programs that are:
  - Understandable, readable, reusable, efficient, easy to develop and debug

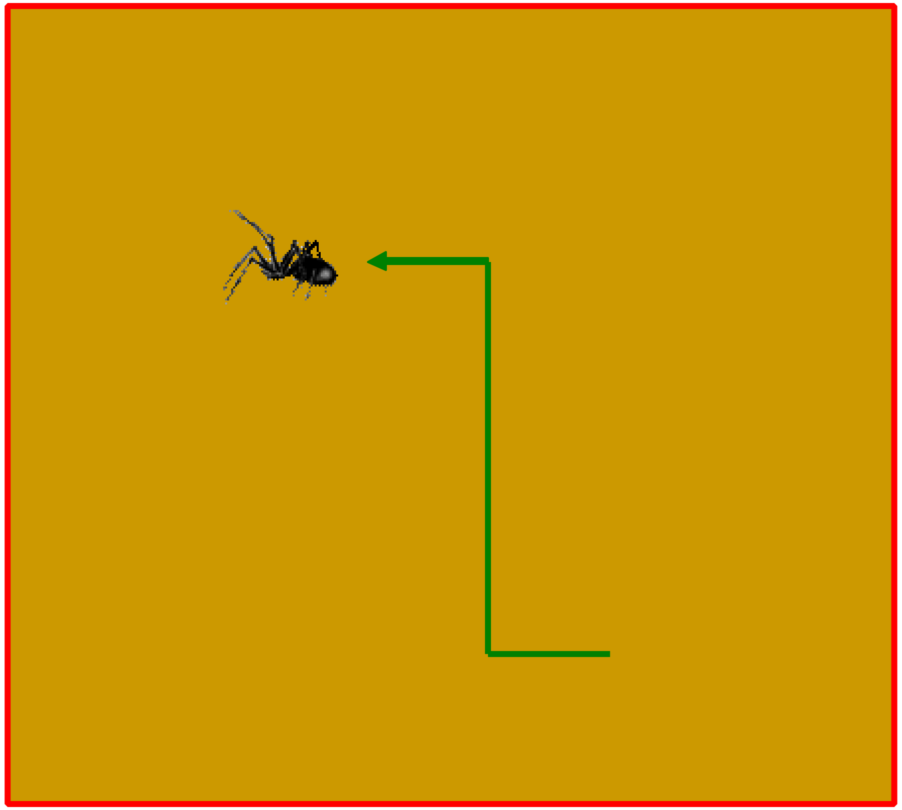
Comments, headers, and programming style

- Comments start with “--” and are ignored by the compiler

```
-------------------------------------
-- program name:  my_first_program
-- programmer: Jane B
-- usage:
-- compile:
-- system:
-- date: started 9/5/03
--          phase 1 complete 9/8/03
-- bugs:
-- description:
------------------------------------
```

Adventures of the Spider

- Simple picture-drawing creature – The Spider



Spider commands with parameters

```
TYPE Directions IS (North, East, South, West);
TYPE Colors IS (Red, Green, Blue, Black, None);
PROCEDURE Face (WhichWay: IN Directions);
   -- Spider.Face(WhichWay => Spider.West);
PROCEDURE ChangeColor (NewColor: Colors);
   -- Spider.ChangeColor(NewColor => Spider.Red);
PROCEDURE Step;
PROCEDURE TurnRight;
PROCEDURE TurnLeft;
```

Algorithm with nested loop

- Algorithm for drawing a box:

```
FOR Side IN 1..4 LOOP
   Spider.ChangeColor(Spider.RandomColor);
   FOR Count IN 1..5 LOOP
      Spider.Step;
   END LOOP;
   Spider.TurnRight;
END LOOP;
```

Run–time error

```
WITH Spider;
PROCEDURE Spider_Crash IS
BEGIN -- Spider_Crash
   Spider.Start;
   Spider.ChangeColor(NewColor => Spider.Red);
   FOR Count IN 1..99999 LOOP
         Spider.Step;
   END LOOP;
   Spider.Quit;
END Spider_Crash;
```

Conditional execution

```
FUNCTION AtWall RETURN Boolean;
-- Pre:   None
-- Post: Return True if the spider is standing
--          next to a wall

IF Spider.AtWall THEN
   EXIT;
END IF;
```

Exercise (1)

- Modify the “Hello” program to display the following text on the screen

Hello World

My name is Your Name

Exercise (2)

- Write an algorithm to use the Feldman "spider package" to draw an inverted triangle as shown below.

```
RRRRRRR
 R     R
  R   R
    R
```

Ada Syntax

General structure of Ada programs

with ...;

-- header

--------------------------------------- -

procedure program_name **is**

  declare constants & variables used

Begin -- program_name

  statements

end program_name;

```
with Ada.Text_Io;
procedure Hello_Name is
-------------------------------------------------------------    -------------------
--|Requests, then displays, user's name
--| Author: Michael Feldman, The George Washington University
--| Last Modified: June 1998
--------------------------------------------------------------------------------
    FirstName: String(1..10); -- object to hold user's name
begin -- Hello_Name
    -- Prompt for (request user to enter) user's name
    Ada.Text_IO.Put
          (Item => "Enter your first name, exactly 10 letters.");
    Ada.Text_IO.New_Line;
    Ada.Text_IO.Put
          (Item => "Add spaces at the end if it's shorter.> ");
    Ada.Text_IO.Get(Item => FirstName);
    -- Display the entered name, with a greeting
    Ada.Text_IO.Put(Item => "Hello ");
    Ada.Text_IO.Put(Item => FirstName);
    Ada.Text_IO.Put(Item => ". Enjoy studying Ada!");
    Ada.Text_IO.New_Line;
end Hello_Name;
```

Modules

- Procedure
  - Abstracts an operation
- Package
  - Collects related operations and data types

Advantages of modules

- Procedures
  - Functional abstraction
  - Top-down development
  - Reduced complexity
  - Parallel development
  - Avoid duplication
- Packages
  - Shared resources
  - Improved productivity
  - Improved quality

Package

- Collection of resources
- Encapsulated in one unit
- Ex: Text_IO, Calendar, user-defined packages
  - Collection of types and constants
  - Group of related subprograms
  - User defined types and allowable operation

Reserved words and identifiers

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

Reserved words and identifiers

- Pre-defined words
  - Boolean Character Close Create Delete False Float **Get Integer** Natural **New_Line** Open **Put** Put_Line Positive Read Reset **Skip_Line** String Text_Io True Write

Layout conventions

- Common layout convention makes programs easier for others to read, understand (and mark!)
- Basic conventions
  - One statement (one thought) per line
  - Break long lines into readable segments
  - Indent lines to show different parts of program
  - Blank lines separate parts of the program
  - Comments help readers understand program

-- Comments

- Minimum comments in any program:
  - the name of the program
  - who wrote it and when
  - description of what the program does
  - description of any constants or variables
  - description of purpose of each segment of code
  - assumptions made (precondition / postcondition)

Types of statements

- Input/Output libraries
  - Text: Ada.Text_Io
  - Integer: Ada.Integer_Text_Io
  - Float: Ada.Float_Text_Io
  - Own type: define new library

Types of statements

- Input/Output libraries

```
type Colors is(white, black, red, purple);
package Color_Io is
   new Ada.Text_Io.Enumeration_Io (Enum => Colors);


One_Color : Colors;
begin -- procedure_name
   Color_Io.Get (Item => One_Color);
```

Types of statements

- Input
- **Get** (argument)
  - Argument is a variable that receives input values
  - Value must be same type (e.g., integer) as variable

Put (Item => “Please enter the first number: “);

Get (Item => Number1);

Types of statements

- **Skip_Line**
  - Advance to next line, ignoring unused input

Put (Item => “Please enter the first number “);
Get (Item => Number1); Skip_Line;
Put (Item => “Please enter the second number “);
Get (Item => Number2); Skip_Line;

Please enter the first number 42 10
Please enter the second number 23

Types of statements

- Output
- **Put** (argument)
  - Print argument
  - Leave the cursor on the same line

Put(Item => “Please enter the first number: “);

Get(Item => Number1); Skip_Line;

Please enter the first number: 42

Types of statements

- Formatted output

```
Put ("The sum of the numbers is:");
Put (Number1+Number2, Width=>7); New_Line;
Put ("The product of the numbers is:");
Put (Number1*Number2, Width=>3); New_Line;
Put ("The sum of the numbers is:");
Put (Number1+Number2, Width=>1); New_Line;
```

```
The sum of the numbers is:     14
The product of the numbers is: 48
The sum of the numbers is:14
```

Types of statements

```
Put (23.456);
Put (23.456, Exp=>0);
Put (23.456, Aft=>3, Exp=>0);
Put (23.456, Aft=>2, Exp=>0);
Put (23.456, Fore=>3, Aft=>3, Exp=>0);
```

```
' 2.34560000000000E+01'
'23.45600000000000'
'23.456'
'23.46'
' 23.456'
```

Types of statements

- Assignment
  - Perform calculation and save result in a variable

Total_Num := Number1 + Number2;

Data types

- A **variable** has a
  - Name
    - An Identifier
    - What does the variable **represent**?
  - Data type
    - What **values** can the variable have?
    - What **operations** can be performed on it?

Data types

- Main pre-declared data types in Ada
  - Integer
  - Float
  - Character
  - String
  - Boolean

Data types

- **Constants** are data values that does not change
  - **Name : constant Type := Value;**

Answer : **constant** String := "forty two";

Medicare_Rate : **constant** Float := 1.4;

Pi : **constant** Float := 3.1415926536;

Data types

- Ada has **strong typing**
  - 3 + 4
  - 3.0 / 4.0
  - 1.0 > 0
  - 3 * 4.0
- Mixed arithmetic: must convert one type to another
  - 1.0 > FLOAT(0)
  - FLOAT(3) * 4.0
  - 3 * INTEGER(4.0)

Integer type

- Positive or negative number with **no** decimal part
  - 354 -52689 +4432
- Range of integers
  - **Integer'First** :smallest integer on given system
  - **Integer'Last** :largest integer on given system

Put ("The lowest integer value is: ");

Put (Integer'First); New_Line;

Integer type

- arithmetic
  - unary minus (negation) **-int_val**
  - absolute value **abs int_val**
  - **+ - * / mod rem ****

division	**23 / 4 = 5**
remainder	**23 rem 4 = 3**
modulus	**-23 mod 4 = 1**
exponentiation	**2 ** 4 = 16**

- relational:
  - **= /= < > <= >=**

Exercise (1)

- Write an algorithm to
  - a. Accept the weight of the user (in kilograms)
  - b. Compute the equivalent weight in pounds
  - c. Display:

  "weight_in_kg" kg = "weight_in_pounds" lb

  - **Hint**: 1 pound = 0.453592 kilograms.

String type

- Used when representing a sequence of characters as a single unit of data
  - How many characters?
  - String (1 .. Maxlen);

```
Max_Str_Length : constant := 26;
Alphabet, Response:String(1..Max_Str_Length);
```

String Operations

- Assignment

```
Alphabet := “abcdefghijklmnopqrstuvwxyz”
Response := Alphabet;
```

- Concatenation (&)

```
Alphabet(1..3) & Alphabet(26..26)

Put(Item => “The alphabet is “ & Alphabet & “.”);
```

Sub–strings

- Individual character: specify position

alphabet(10)   'j'

alphabet(17)   'q'

- Slice: specify range of positions

alphabet(20..23)   "tuvw"

alphabet(4..9)   "defghi"

- Assign to compatible slice

response(1..4) := "FRED";

String I/O

- Text_Io
  - Output: Put, Put_Line
  - Get
    - Exact length needed

    Get(Item => A_String);

  - Get_Line
    - Variable length accepted
    - Returns string and length

    Get_Line(Item => A_String, Last => N);

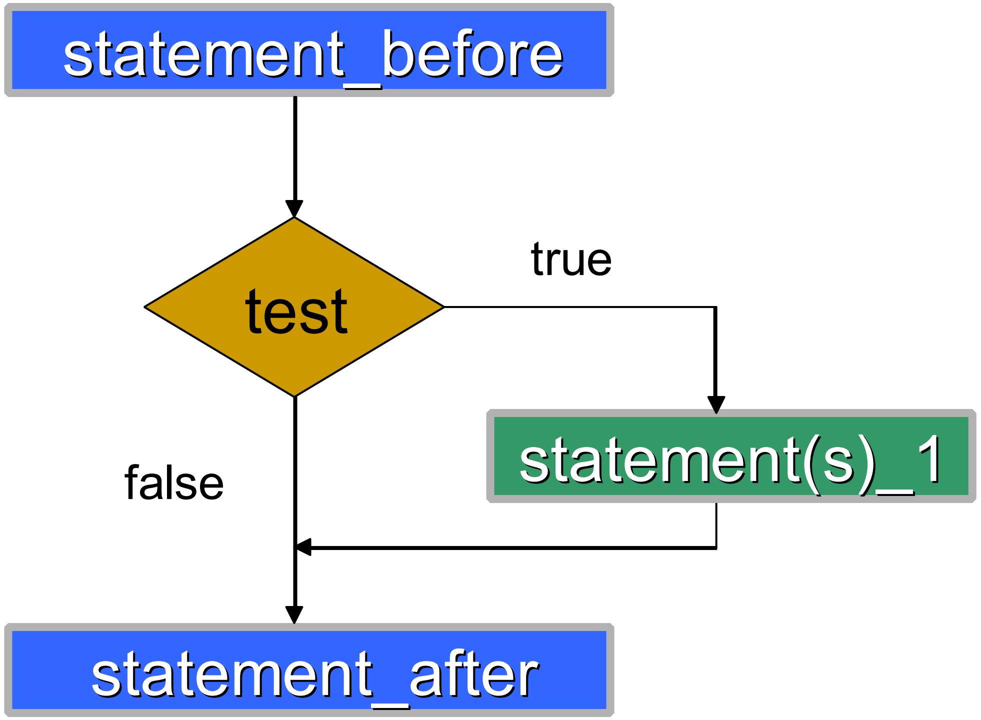
Control Structures
Selection statements

- Ada provides two types of selection statements
  - IF statements
    - **if-then**, when a single action might be done
    - **if-then-else**, to decide between two possible actions
    - **if-then-elsif**, to decide between multiple actions
  - Case statements
    - also for deciding between multiple actions

if–then Statements

- Statement form

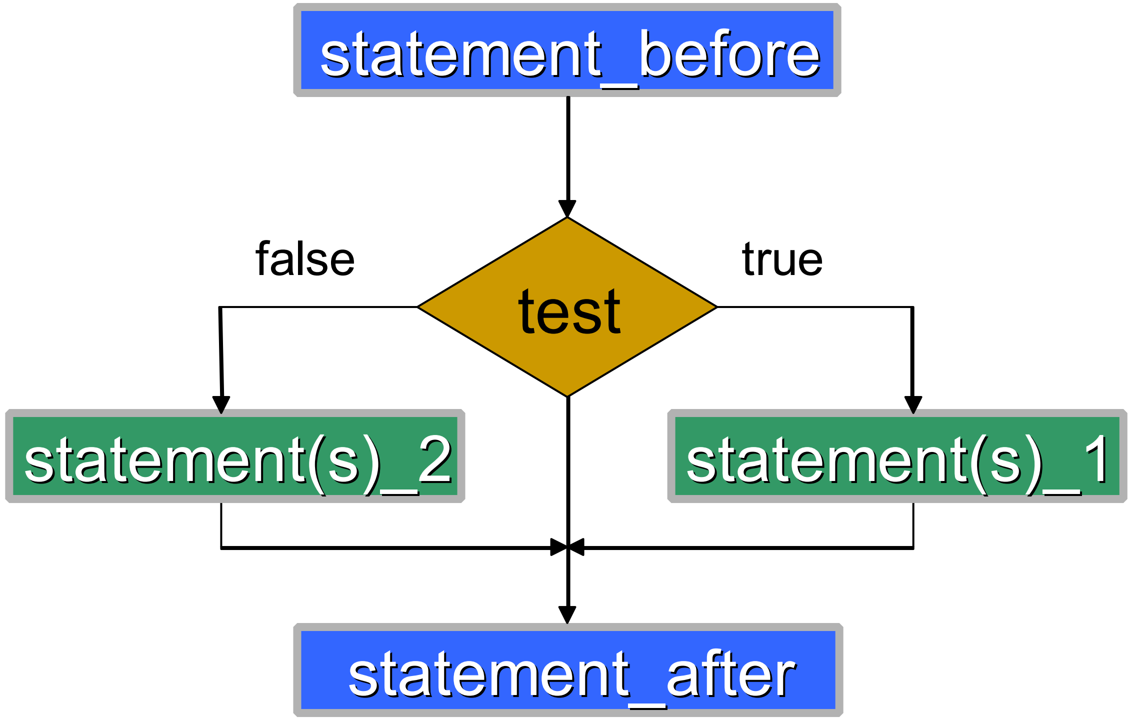
statement_before;
if test **then**
   statement(s)_1;
end if;
statement_after;



if-then-else Statements

- Statement form

statement_before;
if test **then**
    statement(s)_1;
else
    statement(s)_2;
end if;
statement_after;



Multiple Selections

statement_before;
if test_1 **then**
  statement(s)_1;
elsif test_2 **then**
  statement(s)_2;
else
  statement(s)_3;
end if;
statement_after;

if_then_elsif Example

- bank.adb

Balance after withdrawal	**Action**
>= 0	Accept withdrawal
>= -50 and < 0	Overdraft
< -50	Refuse withdrawal

if_then_elsif Example

- Alternative user interfaces

Enter balance of the account **100**

Enter the withdrawal **50**

Accepted. Balance is 50

Enter balance of the account **76**

Enter the withdrawal **150**

Refused! Balance is 76

Enter balance of the account **50**

Enter the withdrawal **75**

Overdraft! Balance is -25

if_then_elsif Example

- Algorithm

1. Get balance and withdrawal
    1. Get balance
    2. Get withdrawal
2. Calculate resulting balance
    1. New balance = old balance – withdrawal
3. If new balance is >= zero

    then

    1. Indicate transaction accepted

    else if new balance between zero and overdraft limit

    2. Indicate overdraft is used

    else

    3. Indicate transaction rejected

if_then_elsif Example

- Data design

NAME	TYPE	Notes
Overdraft_Limit	Integer	-50 (for ease of change)
Zero	Integer	0 (for readability only)
Balance	Integer	Balance in the account
Withdrawal	Integer	Amount requested by user
Resulting_Balance	Integer	Balance after withdrawal

Conditions

- NOT

NOT(TRUE)	FALSE
NOT(FALSE)	TRUE

- OR

F or F	F
F or T	T
T or F	F
T or T	T

Conditions

- AND
- XOR

F and F	F
F and T	F
T and F	F
T and T	T

F xor F	F
F xor T	T
T xor F	T
T xor T	F

Conditions Examples

- (age < 18) **or** (sex = ‘F’)
- **not** ((age >= 18) **and** (sex = ‘M’))
- ((age >= 55) **and** (sex = 'F')) **or** ((age >= 60) **and** (sex = 'M'))

Loop Statements

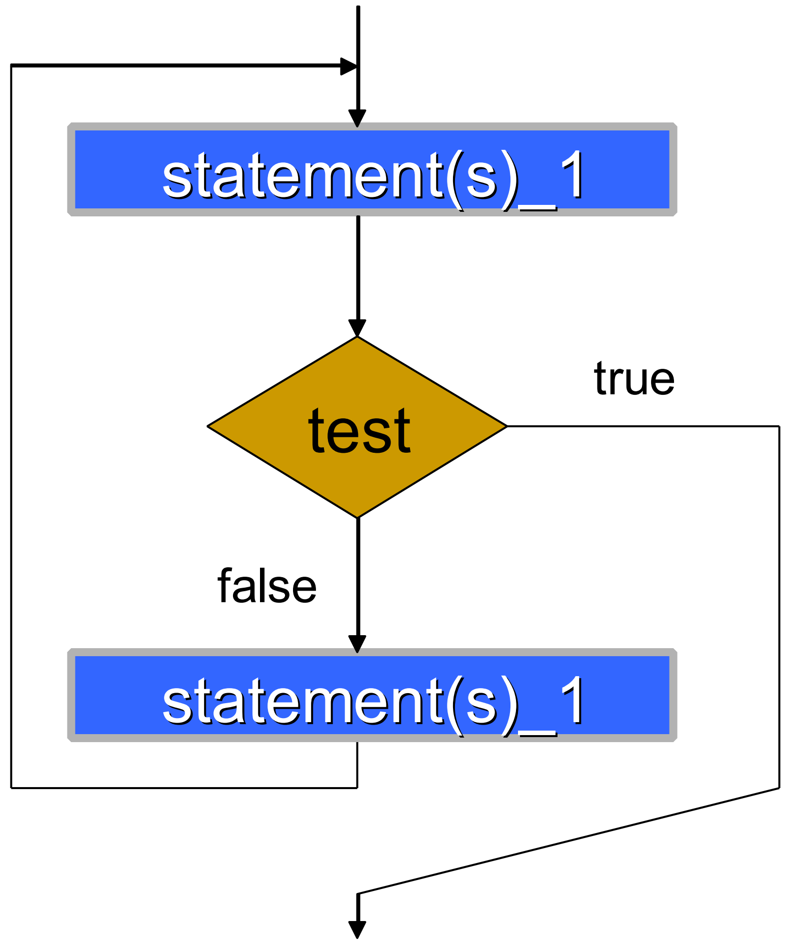
- **Definite iteration**
  - where the set of actions is performed a known number of times. The number might be determined by the program specification, or it might not be known until the program is executing, just before starting the iteration.
  - Ada provides the **FOR statement** for definite iteration.

Loop Statements

- **Indefinite iteration**
  - where the set of actions is performed a unknown number of times. The number is determined during execution of the loop.
  - Ada provides the **WHILE statement** and general **LOOP statement** for indefinite iteration.

General Loop Statements

loop
  statements_1;
  exit when test;
  statements_2;
end loop;



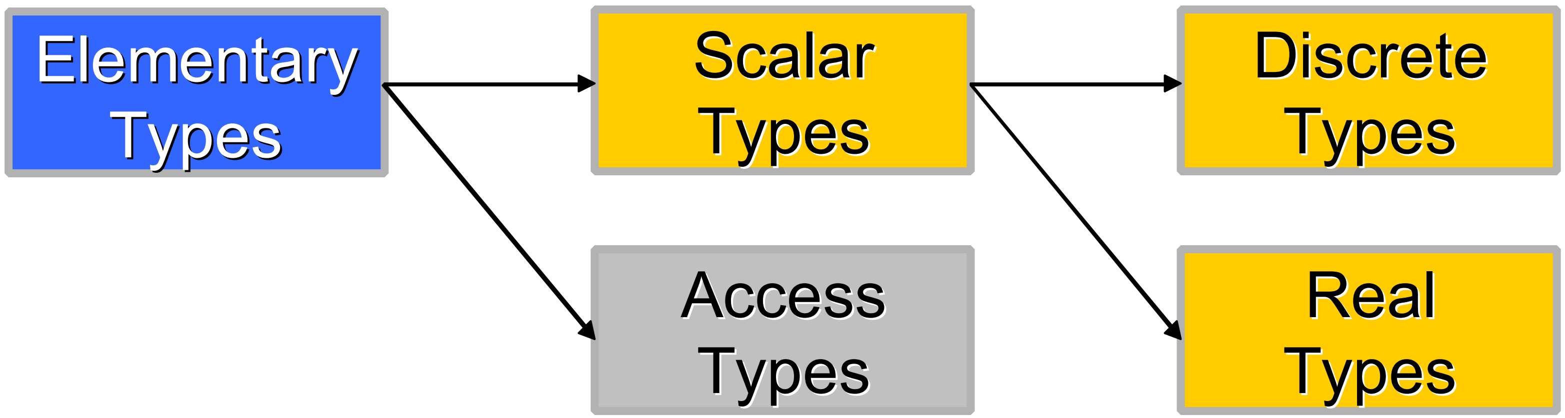
Types

Types

- Type
  - A set of **values**
  - A set of **primitive operations**
- Grouped into classes based on the similarity of values and primitive operations
  - **Elementary** types
  - **Composite** Types

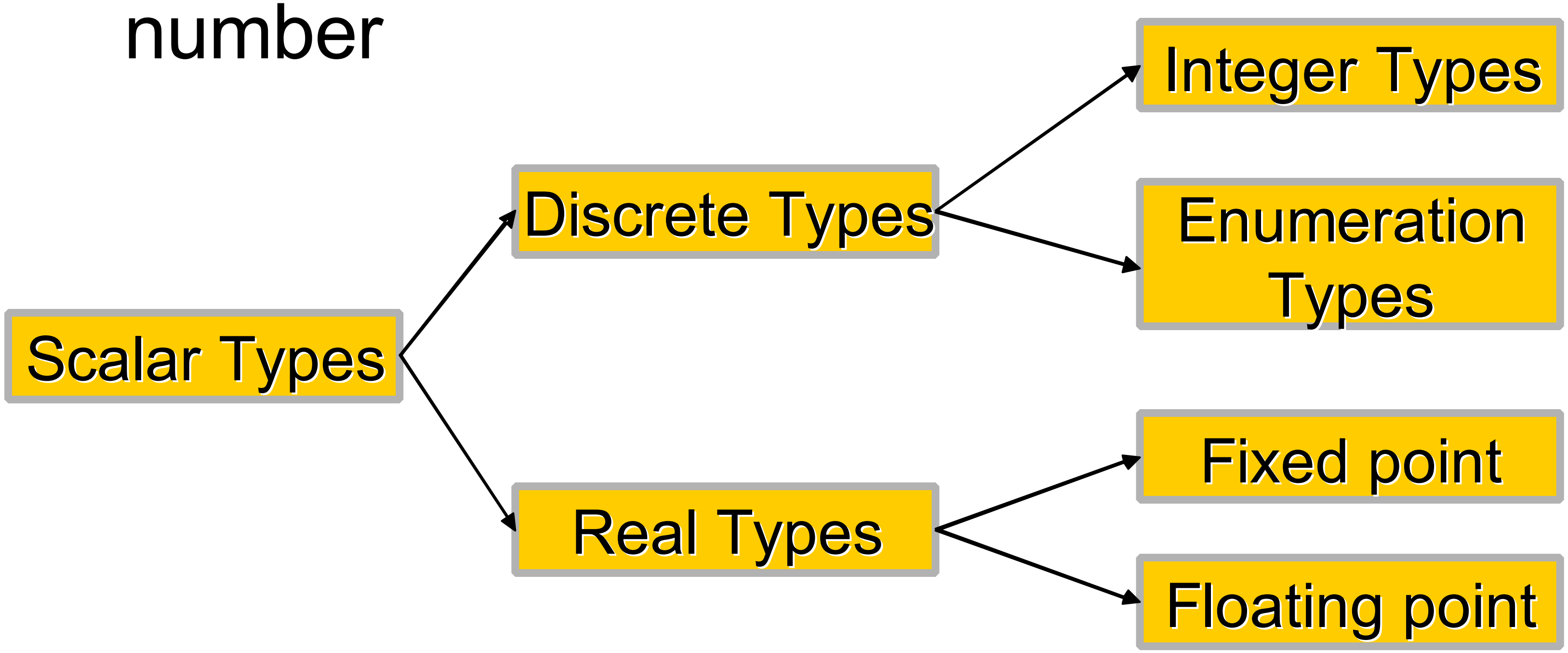
Type Classification

- **Elementary** Types : Values are logically indivisible
- **Composite** Types : Values composed from components



Scalar Types

- Ordered → relational operators are defined
- Each value of a discrete type has a position number



Attributes of Scalar Types

- S'First denotes the lower bound of the range of S. The value of this attribute is of the type of S.
- S'Last denotes the upper bound of the range of S
- S'Range is equivalent to the range S'First .. S'Last

Operations on Scalar Types

- S'Min returns lower of two elements
- S'Max returns higher of two elements
- S'Value accepts a string and returns the value in the type
- S'Image converts the value into a string
- S'Pred and S'Succ – behavior depends on the scalar type
  - S'Pred (Integer) : returns (Integer -1)
  - S'Succ (Integer) : returns (Integer + 1)

Subtypes

- A **subtype** is a subrange of a larger type.
- Subtypes of the same larger type are *not* distinct types. A subtype and the larger type are also not distinct types. Thus subtypes of the same thing are assignment-compatible.
- The benefit of subtypes is that range checks avoid some nonsense.

Subtype Example

- Two useful sub-types of the integers are built into Ada:
  - **subtype** POSITIVE **is** INTEGER **range** 1..INTEGER'LAST;
  - **subtype** NATURAL **is** INTEGER **range** 0..INTEGER'LAST;

Subtype Example

```
min_on_bus :       constant := 0;
max_on_bus :       constant := 80;
type no_on_buses is range min_on_bus ..
   max_on_bus;

max_seated : constant no_on_buses := 50;

subtype seated_on_buses is no_on_buses range
   min_on_bus .. max_seated;
subtype standing_on_buses is range min_on_bus ..
   (max_on_bus - max_seated);
```

Subtypes

subtype Natural **is** Integer **range** 0..Integer'Last;
subtype Positive **is** Integer **range** 1..Integer'Last;
subtype NonNegativeFloat **is** Float **range** 0.0 .. Float'Last;
subtype SmallInt **is** Integer **range** -50..50;
subtype CapitalLetter **is** Character **range** 'A'..'Z';

X, Y, Z : SmallInt;
NextChar : CapitalLetter;
Hours_Worked : NonNegFloat;

X := 25;
Y := 26;
Z := X + Y; -- Exception raised

Operations on Discrete Types

- S'Pos(Arg) returns the position number of the argument
- S'Val(Arg) a value of the type of S whose position number equals the value of S

Enumeration Types

- A data type whose values are a collection of allowed words

type Class **is** (Freshman, Sophomore, Junior, Senior);

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

type colours **is**

(white, red, yellow, green, blue, pink, black);

type traffic_colours **is** (green, yellow, red);

type suits **is** (clubs, diamonds, hearts, spades);

Enumeration Types

- Enumeration types have the following benefits:
  - readable programs
  - avoid arbitrary mapping to numbers, e.g. better to use "Wed" than 3 for a day of the week
  - they work well as selectors in case statements

Attributes of Enumerated Types

type Days **is** (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);

Today : Days; --current day of the week

Tomorrow : Days; --day after Today

Today := Friday;

Tomorrow := Saturday;

- Days'First is **Monday**
- Days'Last is **Sunday**
- Days'Pos(Monday) is **0**
- Days'Val(0) is **Monday**
- Days'Pred(Wednesday)is **Tuesday**
- Days'Succ(Today) is **Saturday**

You must ensure the result is legal. A **CONSTRAINT_ERROR** will occur at run-time otherwise. For example, **days'SUCC(Sun)** is illegal.

Derived Types

- age := -20;
- height := age - class_size;
- shoe_size := 2 * no_on_bus;

- Types help program values reflect the real world.

Derived Integer Types

- New data types can be **derived** from INTEGER:

type ages **is new** INTEGER **range** 0 .. 110;
   age : ages;
   voting_age : **constant** ages := 18;
type heights **is range** 0 .. 230;
   height : heights;

min_enrolment : **constant** := 6;
max_enrolment : **constant** := 200;
type class_sizes **is range** 0..max_enrolment;
class_size : class_sizes;

Type conversion

- Ada has *strong typing:* different types cannot be mixed
- Explicit type conversion is permitted:

```
type length is digits 5 range 0.0 .. 1.0E10;
type area is digits 5 range 0.0 .. 1.0E20;
function area_rectangle (L,H : length) return area is
begin
   return area(L) * area(H);
end;
```

Benefits of derived types

- Nonsense rejected by compiler
  - height := age - class_size;
- "Out of range" rejected by compiler
  - age := -20;
- “Out of range” run time error
  - class_size := class_size + 100;
- Enforce distinct nature of different objects
- Robust, elegant, effective programs

I/O Libraries

- Each distinct type needs its own I/O library.
- General form:

```
package type_io is new
            TEXT_IO.basetype_io (typename);
```

I/O Libraries

- **package** int_io **is new** TEXT_IO.INTEGER_IO (INTEGER);

- **type** ages **is new** INTEGER **range** 0 .. 110;
- **package** ages_io **is** new TEXT_IO.INTEGER_IO (ages);

- **type** measurement **is** digits 10;
- **package** measurement_io **is new** TEXT_IO.FLOAT_IO (measurement);

- **type** suits **is** (clubs, diamonds, hearts, spades);
- **package** suits_io **is new** TEXT_IO.ENUMERATION_IO (suits);

- **type** colours **is** (white, red, yellow, green, brown, blue, pink, black);
- **package** colours_io **is new** TEXT_IO.ENUMERATION_IO (colours);

Input/Output Operations

```
type Days is (Monday, Tuesday, Wednesday,
   Thursday, Friday, Saturday, Sunday);
package Day_IO is new
   Ada.Text_IO.Enumeration_IO(Enum=>Days);

Day_IO.Get(Item => Today);
Day_IO.Put(Item => Today, Width => 10);
if Today in weekend_days then
   put(”Holliday!”);
end if;
```

Exercise (1)

- Write an Ada95 program to accept a date in the **Date/Month/Year** format. Accept each of the inputs separately. Display the date in all three formats as shown below:
  - 19/9/2003
  - 19 September 2003
  - 19.IX.2003

Exercise (2)

- What are the First and Last values of the following data types:
  - Integer
  - Float
  - Character
  - Boolean

Structuring Programs

Structuring Programs

- Mechanisms to control complexity
  - Abstraction
  - Modularization
  - Encapsulation

Structuring Programs

- Modularity
  - Partition system into modules
    - Reduce complexity, easier development/maintenance
    - Parallel development, divide programming job for teams
  - Ada modules
    - Subprograms: **procedures**, **functions**
    - **Packages**

Procedures

declaration

definition

procedure name (arguments) **is**

-- comment explaining purpose

constant definitions
variable definitions
procedure definitions

defined for use in this procedure only

begin --name
statements
end name;

Example

```
procedure display is
    -- display a number
    num : integer;
    begin -- show_answer
        num:= 71;
        new_line;
        put(“the number of students is :”);
        put(num);
        new_line;
    end display;
```

Procedure Call

- Write its name
- Include arguments in brackets

```
begin
    get_two_nums;
    add_two_nums;
    show_answer;
end;
```

procedure calls

- Procedure must be **visible**
  - Declared earlier
  - Included via **with**

Procedure Call and Return

- Procedure call
  - Remember where we are in **calling** code
  - Transfer to **called** procedure
    - Set up storage for local variables
    - Associate parameters with values
    - Start execution at first statement of callee

Procedure Call and Return

- Procedure finishes executing
  - Wind up **called** procedure
    - Return value through parameter
    - Dispose of storage
  - Pick up where left of in **caller**

Functions

- Effect is to compute a single result
- Returns the result directly
- Function definition: like procedure, except
  - *function* instead of *procedure* as first word
  - Define data type of returned value
  - Include statements to return a value

  return value;
  - Causes immediate termination of the function
  - There cannot be an execution path through the function that does not include a return statement

Example

```
------------------------------------------
-- abs: absolute value
------------------------------------------
function abs (x : in INTEGER) return INTEGER is

begin -- abs
            if x >= 0 then
                        return x;
            else
                        return -x;
            end if;
end abs;
```

```
y := abs (x);
y := 10 * abs (-4);
y := abs (10 - abs (x));
```

Procedures with Parameters

- Parameters (argument to a procedure)
  - The procedure **declaration** shows the number and type of arguments
    - Formal parameter
  - The procedure **call** supplies specific arguments
    - Actual parameter
- Parameter modes
  - Indicate how data may be communicated between calling and called procedure

Formal Parameters

- Procedure declaration defines **formal** parameters
  - general rules for every call to procedure
    - Mode: **in**, **out**, **in out**
    - data type: integer, character, ...
    - internal name: (for use inside procedure)
  - In brackets after procedure name

Formal Parameters

```
procedure adjust (
    exam : in INTEGER;           -- exam mark
    mark : in out INTEGER;       -- overall subject mark
                  )
is
    -- local declarations
begin
    -- statements
end adjust;
```

Actual Parameters

- procedure **call** includes **actual parameters**
  - *specific* parameter values for *this* call

```
begin
   get_exam (exam);
   get_lab (labs);
   mark := exam + labs;
   adjust (exam, mark);
   PUT (mark);
   print_grade (mark);
end;
```

Function or Procedure?

```
-----------------------------------------
-- abs: absolute value
-----------------------------------------
procedure abs (x : in INTEGER; -- argument
               y :out INTEGER; -- abs(argument)
              ) is
begin -- abs
        if x >= 0 then
                y := x;
        else
                y := -x;
        end if;
end abs;
```

```
abs(x, y);              -- y := abs (x);

abs (-4, temp);         -- temp:= abs(-4);
y := 10 * temp;         -- y := 10 * abs (-4);
```

Parameter Modes

- Named from perspective of called procedure
  - **in** supplied to procedure by its caller
  - **out** provided by procedure to its caller
  - **in out** supplied to procedure by caller, (possibly) modified, and handed back

Exercise (1)

- Write a simple package that contains
  - A type definition
  - Two functions called Successor and Predecessor that work exactly like Type'Succ and Type'Pred. The only difference being:
    - Successor(Type'Last) = Type'First
    - Predecessor(Type'First) = Type'Last

Scope of Declarations

- Where does a given declaration apply?
- What declarations apply at a given point?
- Scope of a declaration
  - From where it is made, to the end of the subprogram that contains it

Visibility

```
procedure P is
   X : Integer;
   procedure Q is
      X : Integer; -- hides outer declaration
   begin
      X := 2; -- local decl. directly visible
      P.X := 3; -- global decl. Visible, but not directly
   end Q;
begin
   Q;
end;
```

A declaration can be hidden from direct visibility, but not hidden from all visibility, and can be accessed using selector syntax

Visibility

Some declarations are hidden from all visibility, in particular once an inherited declaration is overridden, there is no way to name it:

type T **is** (A, B, C, D);

procedure P (X : T);

type T1 **is new** T;

-- inherited P is visible

procedure P (X : T1);

-- inherited P is hidden from all visibility

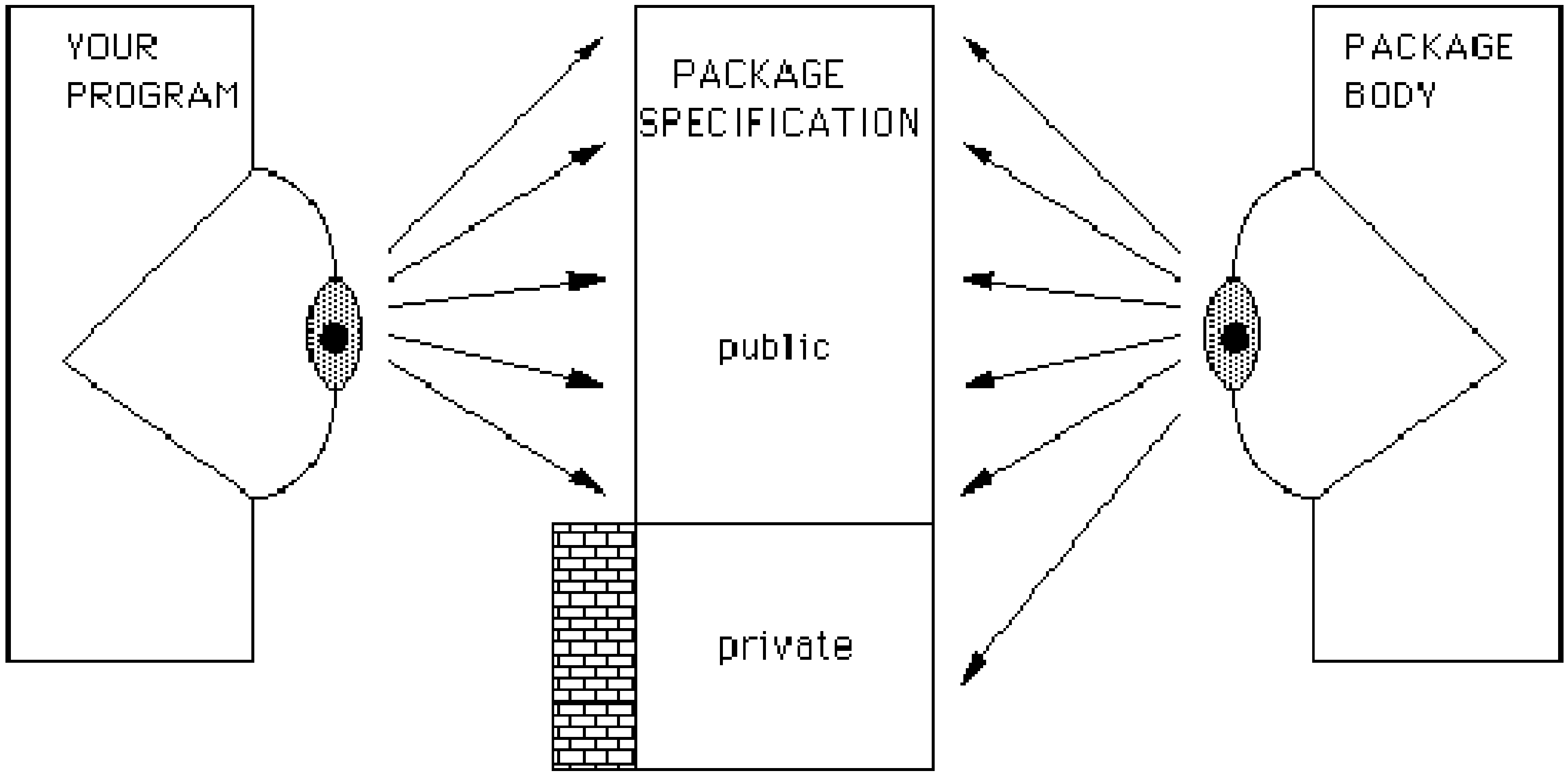
Packages

- Collection of resources
- Encapsulated in one unit
- Single library unit
  - Free-standing unit
    - Must contain its own declarations for everything it needs
  - Compiled on its own
    - Incorporated in other programs via ‘with’
    - Compilation order:
      - Library unit
      - Procedures that use it

Package Organization

- Package **specification** show “what” it provides
- Package **body** defines “how” it is implemented
- Both are separate from the user’s program that uses the package

Package Organization



Courtesy of Chris Lokan. Used with permission.

Package Specification

package package_name **is**

declarations

public portion

private

type definitions

end package_name;

private portion

Package Specification

- Public:
  - What you need to know to use the package
- Private:
  - Implementation of data types

Private Types

```
package accounts is
    type account is private; -- declaration comes later
procedure withdraw(an_account : in out account; amount : in money);
procedure deposit(an_account: in out account; amount : in money);
function create(initial_balance : money) return account;
function balance( an_account : account) return integer;

private  -- this part of the package specification contains the full description.
type account is
    record
            account_no :positive;
            balance :integer;
    end record;
end accounts;
```

Package Body

- Implementation of the resources provided by the package
- All a user of the package needs to know is what the package provides.
- The package is a "black box" to the user of the package.
- The package body is not visible to a package user.

Package Body

```
package body package_name is
   declarations
end package_name;
```

Package Example specification .ads

```
package PLANIMETRY is

   type length is digits 5 range 0.0 .. 1.0E10;
   type area is digits 5 range 0.0 .. 1.0E20;

   function area_rectangle (L,H : length) return area;
   function area_circle (R : length) return area;
   function area_triangle (B,H : length) return area;
   function circumf_circle (R : length) return length;

end PLANIMETRY;
```

Package Example body .adb

```
package body PLANIMETRY is

   PI : constant := 3.1415926536;

   function area_rectangle (L,H : length) return area is
   begin
         return area(L) * area(H);
   end;

   function area_circle (R : length) return area is
   begin
         return PI * area(R) ** 2;
   end;
```

Package Example body .adb

```
    function area_triangle (B,H : length) return area is
    begin
          return area(B) * area(H) / 2.0;
    end;

    function circumf_circle (R : length) return length is
    begin
          return 2.0 * PI * R;
    end;

end PLANIMETRY;
```

Using Packages

- To use a package element
  - package.element
- Example:

```
Ada.Text_Io.put (item => "abc");
Ada.Text_Io.new_line;
int_io.put (mark, width => 1);
planimetery.area_circle (2.0);
```

Using Packages

- USE allows package to be omitted

```
use Ada.Text_io, int_io, planimetry;
…
put ("abc");
new_line;
put (mark, width => 1);
area_circle (2.0);
```

User Program

```
with TEXT_IO, PLANIMETRY;
procedure main is
    use TEXT_IO;
    ... declarations
L : PLANIMETRY.length;          -- length
H : PLANIMETRY.length;          -- height
A : PLANIMETRY.area;            -- area
R : PLANIMETRY.length;          -- radius
begin
    R := ... ;
    A := PLANIMETRY.area_circle (R);
end main;
```

Case Statement

- Used for multiple selections
  - Alternative to multiple if
  - Used when we can explicitly list all alternatives for one selector

Case Statement

```
statement_before;

case selector is
        when value_list_1 =>
                statement(s)_1;
        when value_list_2 =>
                statement(s)_2;
        …
        when others =>
                statement(s)_n;
end case ;

statement_after;
```

Selectors

- Variable or expression resulting in a discrete value
- Selector value_list may be:
    - A single constant value, e.g., ‘a’
    - A series of alternatives, e.g., ‘a’ | ‘b’ | ‘c’
    - A range of values e.g., ‘a’ .. ‘z’
    - Or any combination of the above

Restrictions on Case Statements

- A particular value may only occur **once** in a case statement
- All possible values of the selector **must** be supplied, either explicitly or using **when others**.
- **when others** indicates the action when none of the listed **when** alternatives are matched
  - it must be the last alternative
  - to specify no action, use the "null;" statement

Case vs Multiple if

- Case
  - Table of values and actions
  - Easy to operate specify range of selector values
  - Easy to specify alternative selector values
- Multiple if
  - Sequence of decisions and actions
  - Used when cannot specify range directly
    - Selector is not discrete
    - Choice depends on more than one selector

Exercise (1)

- Explain the difference between the statements on the left and the statements on the right below. For each of them, what is the final value of X if the initial value of X is 3?

```
IF x >= 0 THEN
        x := x + 1;
ELSIF x >= 1 THEN
        x := x + 2;
END IF;
```

```
IF x >= 0 THEN
        x := x + 1;
END IF;
IF x >= 1 THEN
        x := x + 2;
END IF;
```

Exercise (2)

- Write a simple package that contains
  - A function to add two integers.
  - A procedure to multiply two integers.

- Write an Ada95 program that uses the package

Iteration

- Definite iteration
  - FOR statement
- Indefinite iteration
  - WHILE statement
  - General LOOP statement

For Statement

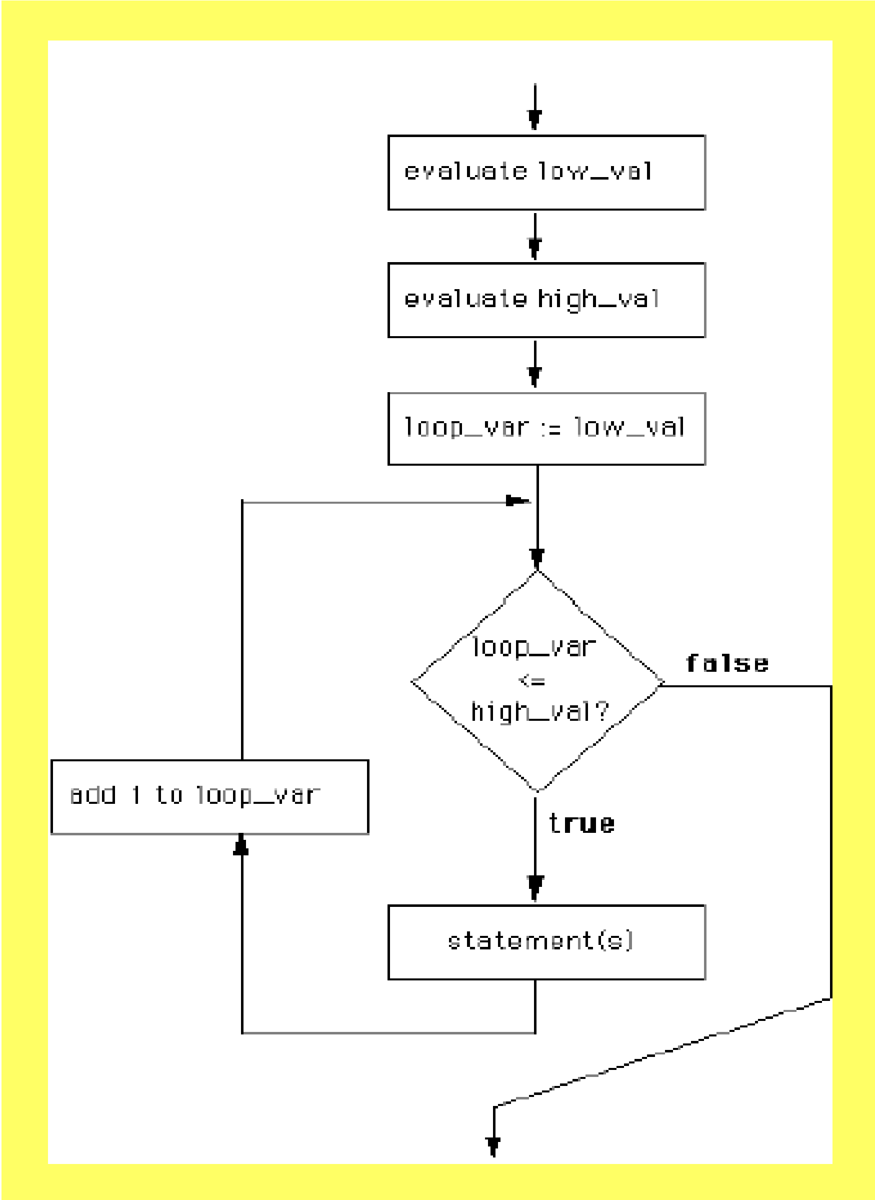
```
for loop_var in low_val .. high_val
loop
   statement(s);
end loop;
```

For Statement

```
for i in -1 .. 10 loop
   PUT(i); NEW_LINE;
end loop;

for i in 1 .. 10 loop
   PUT(i); NEW_LINE;
end loop;

for i in 2 .. n-1 loop
   PUT(i); NEW_LINE;
end loop;
```



Courtesy of Chris Lokan. Used with permission.

WHILE Statement

while test **loop**
  statement(s);
end loop ;

WHILE Statement

- While loops may be designed as a repeat structure, to execute at least once

j := -1;

while (j < 0) **loop**

put (“Enter positive j:);

get (j); skip_line;

end loop;

WHILE Statement

- A while loop may not execute at all

```
tot := 0;
PUT("Enter j (-1 to exit): ");
GET(j); SKIP_LINE;
while (j /= -1) loop
   tot := tot + j;
   PUT("Enter j (-1 to exit): ");
   GET(j); SKIP_LINE;
end loop;
PUT("Total is "); PUT(tot); NEW_LINE;
```