Introduction au langage Ada

Hongyang QU

2007

Le nom Ada

- Il a été choisi en l'honneur d'Ada Lovelace (1815 - 1852).

L'histoire du langage Ada

- Le développement d'Ada a commencé au début des années 1980 pour donner Ada 83, en réponse à un cahier des charges établi par le Département de la Défense américaine (DoD).
- Il a été ensuite repris et amélioré au milieu des années 1990, pour donner Ada 95, le premier langage objet standardisé de manière internationale.
- Le langage dispose maintenant d'une révision (mineure) appelée Ada 2005.

Les caractéristiques du langage Ada

- typage statique et fort;
- modularité;
- syntaxe claire et non ambiguë;
- généricité;
- traits temps réel intégrés au langage;
- bibliothèques standardisées, liens avec les autres langages.

La référence du langage Ada

- Programming in Ada 2005 par John Barnes
- Programmer en Ada par John Barnes
- Ada 95: The Craft of Object-Oriented Programming par John English

- http://libre2.adacore.com

Un programme simple

```
with Ada.Text_IO;                          -- 1
procedure Hello is                         -- 2
begin                                      -- 3
    Ada.Text_IO.Put ("Hello world!");      -- 4
    Ada.Text_IO.New_Line;                  -- 5
end Hello;                                 -- 6
```

enregistrer sous “hello.adb”

Compilation (1)

- La première façon
  - gcc -c *hello.adb*
  - gnatbind *hello[.ali]*
  - gnatlink *hello [.ali]*

Compilation (2)

- La deuxième façon
  - gnatmake *hello*[.*adb*]

Les critères d'appellation

- Les lettres et les chiffres et le « underscore » sont permis dans un nom;
- Un mot doit commencer par une lettre
- Les majuscules et les minuscules ne sont pas distinguées

Les mots réservés

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

La disposition d’un programme

```
procedure X [(paramètres)] is
   -- les déclarations sont ici
begin
   -- les instructions sont ici
end X;
```

Les clauses de contexte

- with « les noms de paquetages »
- use « les noms de paquetages »

Ajouter la phrase use

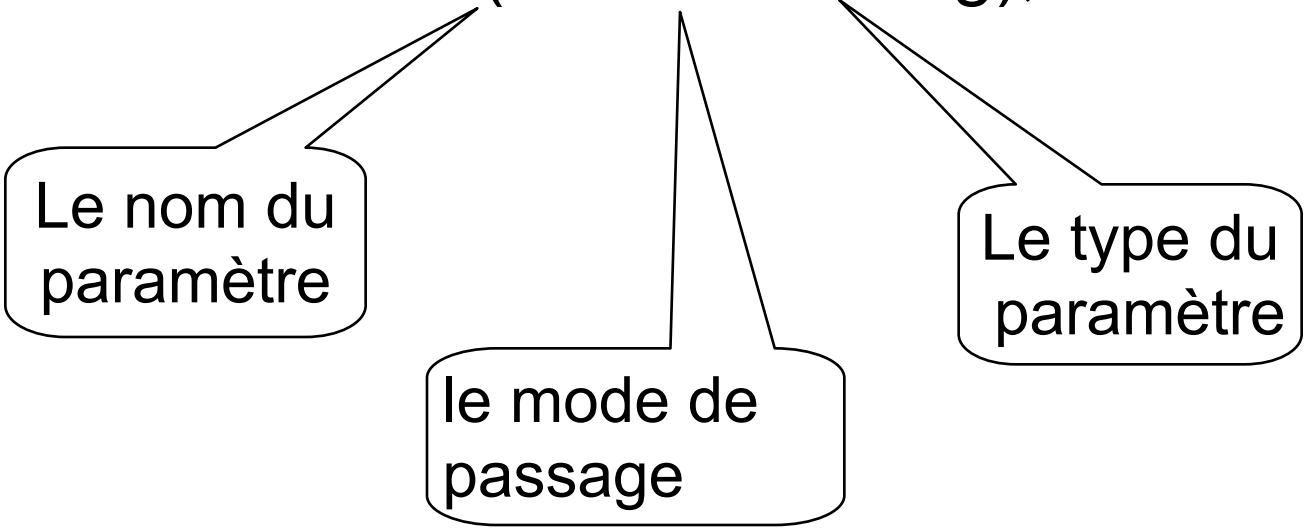
```
with Ada.Text_IO;
procedure Hello is
begin
    Ada.Text_IO.Put ("Hello
   world!");
   Ada.Text_IO.New_Line;
end Hello;
```

```
with Ada.Text_IO;
use Ada.Text_IO;
procedure Hello is
begin
    Put ("Hello world!");
    New_Line;
end Hello;
```

Spécification d'une procédure (1)

- Ada.Text_IO.Put

```
procedure Put (Item : in String);
```



Spécification de la procédure (2)

- Ada.Text_IO.New_Line

```
procedure New_Line (Spacing : in
   Positive_Count := 1);
```

la valeur par défaut

```
New_Line (Spacing => 1);
New_Line (1);
New_Line;
```

```
New_Line (Spacing => 2);
New_Line (2);
```

Spécification de la procédure (3)

- Ada.Integer_Text_IO.Put

```
procedure Put (Item : in Integer;
               Width : in Field := Default_Width;
               Base : in Number_Base := Default_Base);
```

- Put (29); -- " 29"
- Put (29, Width=>1); -- "29"
- Put (29, Width=>5); -- " 29"
- Put (29, Base=>2); -- " 2#11101#" (binaire)
- Put (29, Base=>16); -- " 16#1D#" (hexadécimal)

Put (29, Width=>1,Base=>2) = Put (29, Base=>2,Width=>1) = Put (29,1,2)

Spécification de la procédure (4)

- Ada. Integer_Text_IO.Get

```
procedure Get (Item : out Integer;
               Width : in Field := 0);
```

  - **out** : La procédure n'écrirait qu'au paramètre
  - **in** : La procédure ne lirait que le paramètre
  - **in out** : La procédure écrirait et lirait le paramètre
  - **access** : Il demande un pointeur dont l'adresse ne sera pas modifiée

Instruction d’assignation

- *variable* := *expression*;

- Answer := 'M';
- X := 12;

L'instruction de condition (1)

- **if** *condition* **then**
  instructions
  else
  instructions
  end if;

- **if** *condition1* **then**
  instructions
  elsif *condition2* **then**
  instructions
  elsif *condition3* **then**
  instructions
  ...
  else
  instructions
  end if;

L'instruction de condition (2)

- **case** *expression* **is**
  when *choix* { | *choix*} =>
  instructions
  when *choix* { | *choix*} =>
  instructions
  ...
  when others =>
  instructions

end case;

La condition

- *A* = *B* | *A* /= *B* | *A* < *B* | *A* <= *B* | *A* > *B* | *A* >= *B*
- *condition1* **or** *condition2*
- *condition1* **or else** *condition2*
- *condition1* **and** *condition2*
- *condition1* **and then** *condition2*
- *condition1* **xor** *condition2*
- **not** *condition1*

Des exemples de l'instruction de condition

- ```
  if Answer = 'M' and X > 0 then
      Y := 1;
  else
      if X <= 0 then Y := 0; end if;
  end if;
  ```

- ```
  if Answer = 'M' and X > 0 then
      Y := 1;
  elsif not X > 0 then
      Y := 0;
  end if;
  ```

Des tests du domaine

-
```
case Answer is
    when 'A' .. 'Z' | 'a' .. 'z' =>
        Put_Line ("C'est une lettre!");
    when others =>
        Put_Line ("Ce n'est pas une lettre!");
end case;
```

-
```
if Answer in 'A' .. 'Z' or Answer in 'a' .. 'z' then
    Put_Line ("C'est une lettre!");
else
    Put_Line ("Ce n'est pas une lettre!");
end if;
```

-
```
if Answer not in 'A' .. 'Z' then
    Put_Line ("Ce n'est pas une majuscule!");
end if;
```

L’instruction Null

- **when others =>**
  null;

L'instruction de boucle (1)

- **loop**

  ...

  end loop
- **exit**; | **exit when** *condition*;

- **loop**

  if X=0 then **exit**;

  exit when X=0;

  end loop;

L’instruction de boucle (2)

- **while** *condition* **loop**

  ...

  end loop;

- **while** not X > Y **loop**

  X := X + 1;

  end loop;

L’instruction de boucle (3)

- **for** *test* **loop**

  ...

  end loop;

- **for** N in 1 .. 20 **loop**

  Put (N);

  New_Line;

  end loop;

L'instruction de boucle (4)

- Main_loop:
  loop
  ...
  end loop Main_loop;

- Outer:
  loop
  ...
  Inner:
  loop
  ...
  end loop Inner;
  end loop Outer;

Le traitement de “exception” (1)

```
begin
instructions
exception
    when exceptions =>
        traitement_de_exceptions;
end;
```

Le traitement de “exception” (2)

```
with Ada.Text_TO, Ada.Integer_Text_IO;
use Ada.Text_TO, Ada.Integer_Text_IO;
procedure X is
   Y : Positive;
begin
   Get(Y);
   exception
      when Constraint_Error =>
         Put_Line ("Value out of range");
      when Data_Error =>
         Put_Line ("Error in input -- integer expected");
end X;
```

Le traitement de “exception” (3)

```
loop
    begin
        Put ("Enter an integer: ");
        Get (X);
        exit;
    exception
        when Constraint_Error | Data_Error =>
            Put_Line ("Error in input -- please try again.");
            Skip_Line;
    end;
end loop;
```

Le traitement de “exception” (4)

- **exception**
  when others =>
  instructions;

Le traitement de “exception” (5)

- A := A ** 2;
  B := B ** 2;

- ```
  begin
     A := A ** 2;
  exception when Constraint_Error =>
        Put_Line ("Assignment to A failed")
  end;

  begin
     B := B ** 2;
  exception when Constraint_Error =>
        Put_Line ("Assignment to B failed");
  end;
  ```
  ```

Les types standard de variable

- Boolean (True, False)
- Integer
- Float
- Duration
- Character
- String

Construction de nouveaux types

- **type** heure **is range** 0 .. 23;
- **type** jour **is** (Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche);
- **type** Jour_Ouvrable **is new** Jour **range** Lundi .. Vendredi;

Il crée un type dérivé

- **type** Unsigned_Int **is new** Integer **range** 0 .. Integer'Last;
- **subtype** Natural **is** Integer **range** 0 .. Integer'Last;
- **subtype** Positive **is** Integer **range** 1 .. Integer'Last;

Les opérations entre les types différents

- I : Integer; T : Unsigned_Int;

T := I;	T := 99;	T := Unsigned_Int (I);
I := T;	I := 99;	I := Integer (T);
T := T + I;	T := T + 99;	T := T + Unsigned_Int (I);
T := I + I;	T := 99 + 99;	T := Unsigned_Int (I + 2);

Ada.Integer_Text_IO.Put(T); X

Ada.Text_IO.Put(Unsigned_Int'Image(T)); √

Ada. Integer_ Text_IO.Put(Integer(T)); √

Les opérations entre les types et leurs sous-types

- I : Integer; N : Natural;

```
if I in Natural then
    N := I;
else
    Put_Line ("I can't be assigned to N!");
end if;
```

Les attributs (1)

- First -- la première valeur d'un type
- Last -- la dernière valeur d'un type
- Image(X) -- la transformation d'une valeur X en une chaîne de caractères
- Value(X) -- la transformation d'une chaîne de caractères X en une valeur
- Min(X,Y) -- la plus petite valeur entre les valeurs X et Y
- Max(X,Y) -- la plus grande valeur entre les valeurs X et Y

Les attributs (2)

- Succ(X) -- la valeur suivante d'une valeur X
- Pred(X) -- la valeur précédente d'une valeur X
- Val(X) -- la valeur d'un énumératif d'une position X
- Pos(X) -- la position d'une valeur X dans l'énumération

Des exemples d'attribut

- Ada.Text_IO.Put_Line (Integer'Image (A));
- A = Integer'Value ("98");
- Ada.Text_IO.Put (Integer'Image(Integer'Last));
- B := Jour'Succ(B);
- Ada.Text_IO.Put(Jour'Image(B));
- D := Character'Pos (C);

Les opérations de nombres entiers

+	Addition	-	Soustraction
*	Multiplication	/	Division
rem	Reste(Remainder)	**mod**	Module
**	Exponentiation	**abs**	Valeur absolue

Des exemples d'opérations des nombres entiers

7/5 = 1	7 **rem** 5 = 2	7 **mod** 5 = 2
(-7)/5 = -1	(-7) **rem** 5 = -2	(-7) **mod** 5 = 3
7/-5 = -1	7 **rem** -5 = 2	7 **mod** -5 = -3
(-7)/-5 = 1	(-7) **rem** -5 = -2	(-7) **mod** -5 = -2

Le type du module des nombres entiers

- **type** Time_of_Day **is mod** 86400;
- Un attribut : Time_Of_Day'Modulus => 86400
- Ada.Text_IO.Modular_IO :

```
package Time_Of_Day_IO is new
   Ada.Text_IO.Modular_IO (Time_Of_Day);
```

Les types des nombres réels

- **Les types de virgule flottante**
  type *nom* **is digits** *expression* [**range** *expression* **..** *expression*] **;**

- **Les types de virgule fixe**
  type *nom* **is delta** *expression* **range** *expression* **..** *expression* **;**

- **Les types de décimal**
  type *nom* **is delta** *expression* **digits** *expression* [**range** *expression* **..** *expression*] **;**

Les exemples des types des nombres réels

- **type** My_Float **is digits** 10;
- **type** My_Fixed **is delta** 0.03 **range** 0.0 .. 10.0;
- **type** Decimal **is delta** 0.01 **digits** 12;

Le paquetage des nombres réels

- Ada.Float_Text_IO

```
procedure Put (Item : in Float;
               Fore : in Field := Default_Fore;
               Aft  : in Field := Default_Aft;
               Exp  : in Field := Default_Exp);
```

```
Put (1234.5678);                                  -- " 1.2345678E+003"
Put (1234.5678, Exp=>0);                          -- "1234.5678"
Put (1234.5678, Fore=>5);                         -- "    1.2345678E+003"
Put (1234.5678, Fore=>5, Aft=>2, Exp=>0);         -- " 1234.57"
Put (1234.5678, Fore=>5, Aft=>2);                 -- "     1.23E+003"
```

Des formats de nombres

- 86400 = 86_400 = 864e2
- 3.14159265 = 3.14159_26536 = 31415.926536e–4
- 2#11111# = 16#1F# = 6#51# = 31
- 16#1F#e1 = 1F (= 31) × 16^1 = 496

Des constantes

```
with Ada.Text_IO, Ada.Calendar;
use  Ada.Text_IO, Ada.Calendar;
procedure Greetings is
   Minute : constant           := 60;
   Hour   : constant           := 60 * Minute;
   Noon   : constant Day_Duration := Day_Duration (12 * Hour);
   Start  : constant Day_Duration := Seconds (Clock);
begin
   if Start <= Noon then
      Put_Line ("Good morning!");
   else
      Put_Line ("Good afternoon!");
   end if;
end Greetings;
```

Le type d'énumération

```
with Ada.Text_IO;
use  Ada.Text_IO;
procedure Greetings is
   type Time_Of_Day is (AM, PM);
   package Time_IO is new Enumeration_IO (Time_Of_Day);
   use Time_IO;
   Answer : Time_Of_Day;
begin
   Put ("Is it morning (AM) or afternoon (PM)? ");
   Get (Answer);
   if Answer = AM then
      Put_Line ("Good morning!");
   else
      Put_Line ("Good afternoon!");
   end if;
end Greetings;
```

Les changements de nom

- **package** Latin_1 **renames** Ada.Characters.Latin_1;

- TM : Character **renames**
  Ada.Characters.Latin_1.Registered_Trade_Mark_Sign;

- **procedure** Show (Value : in Float;
  Fore : in Field := 1;
  Aft : in Field := 2;
  Exp : in Field := 0)
  renames Ada.Float_Text_IO.Put;

Des fonctions

- **function** *Nom* [**(** *liste-de-paramètres***)**]
  return *Nom-de-type* **is**

- L'instruction : **return** *expression*;

- Tous les paramètres ont le mode "in".

Les blocs

```
[declare
        -- Les déclarations locales au bloc]
begin
        -- Les instructions
[exception
        -- Le traitement des exceptions]
end;
```

Un exemple

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;
procedure Weekday is
   Day, Month, Year : Integer;
   function Day_Of (Day, Month, Year : Integer) return Integer is
       D : Integer := Day;   M : Integer := Month;   Y : Integer := Year;
   begin
      if M < 3 then   Y := Y - 1; M := M + 10;
      else  M := M - 2;
      end if;
      declare
         C : Integer := Y / 100;
      begin
         Y := Y mod 100;
         return ((26*M - 2)/10 + D + Y + Y/4 + C/4 - 2*C) mod 7;
      end
   end Day_Of;
begin
   Put ("Enter a date: ");  Get (Day);  Get (Month);  Get (Year);
   Put ( Day_Of(Day,Month,Year) );
end Weekday;
```

L'exemple par compilation séparée (1)

- Le fichier weekday.adb

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;
procedure Weekday is
   Day, Month, Year : Integer;
   function Day_Of (Day, Month, Year : Integer)
      return Integer is separate;
begin
   Put ("Enter a date: ");
   Get (Day);
   Get (Month);
   Get (Year);
   Put ( Day_Of(Day,Month,Year) );
end Weekday;
```

L'exemple par compilation séparée (2)

- Le fichier weekday-day_of.adb

```
separate(Weekday)
function Day_Of (Day, Month, Year : Integer) return Integer is
   D : Integer := Day;   M : Integer := Month;   Y : Integer := Year;
begin
   if M < 3 then
      Y := Y - 1; M := M + 10;
   else
      M := M - 2;
   end if;
   declare
      C : Integer := Y / 100;
   begin
      Y := Y mod 100;
      return ((26*M - 2)/10 + D + Y + Y/4 + C/4 - 2*C) mod 7;
   end
end Day_Of;
```

Faire un paquetage (1)

- Le fichier hello_spec.ads

```
package Hello_Spec is
   procedure Hello;
end Hello_Spec;
```

« ce fichier est la spécification du paquetage Hello_Spec. »

Faire un paquetage (2)

- Le fichier hello_spec.adb

```
with Ada.Text_IO;
package body Hello_spec is
   procedure Hello is
   begin
            Ada.Text_IO.Put ("Hello world!");
            Ada.Text_IO.New_Line;
   end Hello;
end Hello_spec;
```

Faire un paquetage (3)

- Le fichier hello_3.adb

```
with Hello_spec;
procedure Hello_3 is
begin
   Hello_spec.Hello;
   Hello_spec.Hello;
   Hello_spec.Hello;
end Hello_3;
```

Des sous-paquetages (1)

- Le fichier greeting.ads

```
package Greeting is
end Greeting;
```

- Le fichier greeting-hello_spec.ads

```
package Greeting.Hello_spec is
   procedure Hello;
end Greeting.Hello_spec;
```

Des sous-paquetages (2)

- Le fichier greeting-hello_spec.adb

```
with Ada.Text_IO;
package body Greeting.Hello_spec is
   procedure Hello is
   begin
            Ada.Text_IO.Put ("Hello world!");
            Ada.Text_IO.New_Line;
   end Hello;
end Greeting.Hello_spec;
```

Des sous-paquetages (3)

- Le fichier hello_3.adb

```
with Greeting.Hello_spec;
procedure Hello_3 is
begin
   Greeting.Hello_spec.Hello;
   Greeting.Hello_spec.Hello;
   Greeting.Hello_spec.Hello;
end Hello_3;
```

Des types de record

- **type** Date_Type **is**
  record
    Day   : Integer;
    Month : Integer;
    Year : Integer;
  end record;

```
Now, Later : Date_Type;

Later.Day   := Now.Day;
Later.Month := Now.Month;
Later.Year  := Now.Year;
Later := Now;

if Later = Now then ...
if Later /= Now then ...

Later := (25,12,1995);
if Now /= (25,12,1995) then ...
Later := (Month=>12, Year=>1995,
          Day=>25);
```

Des chaînes de caractères (1)

- Ada.Text_IO.Put ("Hello world!");
  « Hello world! »

- Ada.Text_IO.Put (""""Hello world!"""");
  « "Hello world!" »

- 'm'

Des chaînes de caractères (2)

- Details : String (1..100);
- Name   : String (1..10);

- subtype Details_Type is String (1..100);
- Details : Details_Type;

- String_1 : String := "Hello world";

- Details(1)   := 'x';
- Details(I+1) := 'x';

- Details(1..5) := "Hello";
- Details(2..6) := Details(1..5);

Des chaînes de caractères (3)

- Name := "0123456789";
- Name := "01234" &
  "56789";
- Details := Details(51..100) & Details(1..50);
- Name := (' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ');
- Name := (1..10 => ' ');
- Name := (others => ' ');
- Details := (1..10 => 'x', 11..13 | 21..23 | 27 | 29 => 'y',
  others => ' ');

- Ada.Text_IO.Get_Line (Details, N);

Des chaînes de caractères (4)

- "abc" = "abc" "abc" /= "ab" "abc" /= ""
- "abc" < "abd" "abc" > "ab" "abc" > ""

- Des tableaux peuvent être comparés par "=", "/=", "<", ">", "<=", ">=" si les différents composants peuvent être comparés.
- Les opérations logiques "and", "or", "not", "xor" sont également définies pour des tableaux si les différents composants supportent ces opérations.

Des tableaux (1)

- **type** CentInteger **is** **array** (1..100) **of** Integer;
- A, B : CentInteger;
- A et B sont comparables.

- A : **array** (1..100) **of** Integer;
- B : **array** (1..100) **of** Integer;
- A et B ne sont pas comparables parce qu'ils ont des types différents.

Des tableaux (2)

- Mois_Longueur : **constant array** (1..12) **of** Positive :=
  (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
- Mois_ Longueur : **constant array** (1..12) **of** Positive :=
  (4 | 6 | 9 | 11 => 30, 2 => 28, others => 31);

- **type** Jour **is**
  (Dim, Lun, Mar, Mer, Jeu, Ven, Sam);
- Demain : **constant array** (Jour) **of**
  Jour := (Lun, Mar, Mer, Jeu, Ven, Sam, Dim);

Des types contraints (1)

- **type** Un_Tableaux **is array** (Positive **range** <>) **of** Integer;
- A : Un_Tableaux (1..100);
- Get (N);

```
declare
   Appts : Un_Tableaux (1..N);
begin
   ...
end;
```

Des types contraints (2)

- First -- la première valeur du type indexé
- Last -- la dernière valeur du type indexé
- Range -- le domaine du type indexé (First .. Last)
- Length -- le nombre d'éléments dans le tableau

```
A : array (1..100) of Integer;
type Jour is (Dim, Lun, Mar, Mer, Jeu, Ven, Sam);
Demain : array (Jour) of Jour := (Lun, Mar, Mer, Jeu, Ven, Sam, Dim);
```

```
A'First = 1
A'Last = 100
A'Range = 1..100
A'Length = 100
```

```
Demain'First = Dim
Demain'Last = Sam
Demain'Range = Dim..Sam
Demain'Length = 7
```

La boucle **for** (1)

- **for** P **in** A'First..A'Last **loop**
  ...
  end loop;

- **for** P **in** A'Range **loop**
  ...
  end loop;

- **with** Ada.Integer_Text_IO; **use** Ada.Integer_Text_IO;

```
procedure Test is
   P : Integer := 0;
begin
   for P in 1..10 loop
      Put(P);        -- la paramètre de boucle P
   end loop;
   Put(P);           -- la variable P déclarée au début
end Test;
```

La boucle **for** (2)

- ```
  with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
  procedure Test is
      I : Integer := 0;
      A : array (1..100) of Integer;
  begin
      for P in 1..100 loop
          I := P;
          exit when A(P) = 0;
      end loop;
      Put(I);
  end Test;
  ```
  ```

La boucle **for** (3)

- **for** P **in reverse** 1..100 **loop** ...
  - P : 100, 99, 98 ...
- **for** P **in** 100..1 **loop** ...
  - La boucle n'est pas exécuté.

Des tableaux de plusieurs dimensions (1)

- Strings : **array** (1..5) **of** String(1..5) :=
  ("SATOR", "AREPO", "TENET", "OPERA", "ROTAS");

- Strings : **array** (1..5, 1..5) **of** Character :=
  (('S', 'A', 'T', 'O', 'R'),
  ('A', 'R', 'E', 'P', 'O'),
  ('T', 'E', 'N', 'E', 'T'),
  ('O', 'P', 'E', 'R', 'A'),
  ('R', 'O', 'T', 'A', 'S'));

Les tableaux de plusieurs dimensions (2)

- **type** File **is** (A,B,C,D,E,F,G,H);
  type Rank **is range** 1..8;
  Board : **array** (File, Rank) **of** Integer;

- Board'Range(1) = A..H Board'Range(2) = 1..8
  Board'First(1) = A Board'First(2) = 1
  Board'Last(1) = H Board'Last(2) = 8
  Board'Length(1) = 8 Board'Length(2) = 8

Des discriminants (1)

- **type** Varying_String (Maximum : Positive) **is**
  record
    Length : Natural;
    Value : String (1..Maximum);
  end record;

- V1 : Varying_String (100);
  V2 : Varying_String (Maximum => 50);
  V1.Length := V2.Maximum;
  ~~V2.Maximum := 100;~~
  subtype Twenty **is** Varying_String (20);
  V3 : Twenty;
  V4 : Varying_String := (Maximum=>20, Length=>5,
  Value=>"Hello ");

Des discriminants (2)

- **type** Varying_String (Maximum : Positive := 80) **is**
  record
    Length : Natural;
    Value : String (1..Maximum);
  end record;

  V1 : Varying_String;
  V2 : Varying_String (100);

- **type** Varying_String (Maximum : Positive := 80) **is**
  record
    Length : Natural := 0;
    Value : String (1..Maximum);
  end record;

  V2 : Varying_String := (Maximum=>8, Length=>5,
                          Value=>"Hello   ");

Des discriminants (3)

- **type** Time_Stamp **is**
  record
    Creation_Time : Ada.Calendar.Time := Ada.Calendar.Clock;
  end record;

Des discriminants (4)

- ```
  type Account_Number is range 1000_0000..9999_9999;
  type Money_Type is delta 0.01 range 0.00 .. 1_000_000.00;
  Last_Account : Account_Number := Account_Number'First;

  function Next_Account_Number return Account_Number is
     New_Account : Account_Number := Last_Account;
  begin
     Last_Account := Last_Account + 1;
     return New_Account;
  end Next_Account_Number;

  type Bank_Account_Type is
     record
        Account : Account_Number := Next_Account_Number;
        Balance : Money_Type     := 0.00;
     end record;
  ```
  ```

Des types limités

- **type** Bank_Account_Type **is**
  limited record
    Account : Account_Number := Next_Account_Number;
    Balance : Money_Type   := 0.00;
  end record;

  John, Fred : Bank_Account_Type;
  ~~John := Fred;~~
  John.Balance := Fred.Balance;

La déclaration de “exception”

- *Nom_exception* : exception;
  - Data_Error : exception renames Ada.IO_Exceptions.Data_Error;
  - Un_Exception : exception;

- **raise** *Nom_exception*;
  - **begin**

```
...
raise Data_Error;
raise Un_Exception;
...
end;
```

La relance des exceptions

- ```
  begin
      ...
  exception
      when Constraint_Error =>
          -- Quelques opérations
          raise Un_Exception;
  end;
  ```

- ```
  begin
      ...
  exception
      when others =>
          -- Quelques opérations
          raise;
  end;
  ```

L’information de “exception” (1)

- **with** Ada.Exception; **use** Ada.Exceptions;

```
begin
    ...
exception
    when Error : Constraint_Error | Data_Error =>
        Put ("The exception was ");
        Put_Line ( Exception_Name(Error) );
end;
```

- Raise_Exception (Une_Exception'Identity, "L’information de l’exception");
- Exception_Message (Une_Exception_Occurrence);

L'information de "exception" (2)

```
with Ada.Text_IO, Ada.Integer_Text_IO, Ada.Exceptions;
use Ada.Text_IO, Ada.Integer_Text_IO, Ada.Exceptions;
procedure TestException is
   A, B : Integer;
begin
   Get(A);    Get(B);
   Put("A/B = ");   Put(A/B);   New_Line;
exception
   when Error : others =>
      Put ("The exception was ");
      Put ( Exception_Name(Error) & " : ");
      Put_Line (Exception_Message (Error));
end;
```

Fichiers (1)

- File : Ada.Text_IO.File_Type;
- Open (File, Mode => Ada.Text_IO.In_File, Name => "diary");
  - In_File, Out_File, Append_File, Inout_File
- Create (File, Name => "diary");
- Close (File);

Fichiers (2)

- Get (File, C);
- Put (File, "xyz");
- Put_Line (File, "xyz");
- New_Line (File);
- Skip_Line (File);

Fichiers (3)

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
procedure Word_Count is
   File    : File_Type;
   Name    : String(1..80);
   Size    : Natural;
   Count   : Natural := 0;
   In_Word : Boolean := False;
   Char    : Character;
begin
```

Fichiers (4)

```
-- Ouvrir un fichier
loop
   begin
      Put ("Enter filename: ");
      Get_Line (Name, Size);
      Open (File, Mode => In_File, Name => Name(1..Size));
      exit;
   exception
      when Name_Error | Use_Error =>
         Put_Line ("Invalid filename -- please try again.");
   end;
end loop;
```

Fichiers (5)

```
-- Traiter le fichier
while not End_Of_File (File) loop
     if End_Of_Line (File) then
          In_Word := False;
     end if;
    Get (File, Char);
    if In_Word and Char = ' ' then
        In_Word := False;
    elsif not In_Word and Char /= ' ' then
        In_Word := True;
        Count   := Count + 1;
    end if;
end loop;
```

Fichiers (6)

```
-- Fermer le fichier
    Close (File);
    Put (Count);
    Put_Line (" words.");
end Word_Count;
```

Les types privés (1)

```
package Dates is
-- La partie visible du paquetage
   subtype Date_Type is Integer range 1..31;
   type Mois_Type is (Jan, Fev, Mar, Avr, Mai, Juin, Juil,
                      Aou, Sep, Oct, Nov, Dec);
   subtype An_Type is Integer range 1901..2099;
   type Date_Type is private;
-- fonctions pour visiter l'intérieur du type privé
   ....
private                    -- La fin de la partie visible
   type Date_Type is
      record
         Jour   : Jour_Type; Mois : Mois_Type; An  : An_Type;
      end record;
end Dates;
```

Des types privés (2)

- Les opérations des types privés :
  - l'assignation **:=**
  - la comparaison d'égalité **=** et **/=**

Des paquetages privés

- **private package** Parent.Enfant **is**
  ...
  end Parent.Enfant;

- Un sous-paquetage ne peut être visité que par le paquetage principal et les sous-paquetages de ce paquetage principal.

Des types limités et privés

- ```
  package Un_Paquetage is
     type Un_Type is limited private; -- l'assignation n'est pas permise
     ...
  private
     -- l'assignation n'est pas permise
     type Un_Type is limited record ... end record;
  end Un_Paquetage;
  ```

- ```
  package Un_Paquetage is
     type Un_Type is limited private; -- l'assignation n'est pas permise
     ...
  private
     type Un_Type is record ... end record; -- l'assignation est permise
  end Un_Paquetage;
  ```

Des constantes retardées

```
package Dates is
   type Date_Type is private;
   Invalide_Date : constant Date_Type; -- constante retardée
   ...
private
   type Date_Type is record
          Jour   : Jour_Type;
          Mois : Mois_Type;
          An  : An_Type;
   end record;
   -- déclaration complète
   Invalide_Date : constant Date_Type := (31,Fev,1901);
end Dates;
```

Des fonctions surchargées

- **function** Sub (Date : Date_Type; Jours : Integer)
  return Date_Type;
- **function** Sub (Premier, Second : Date_Type)
  return Integer;

- **function** "+" (Gauche : Date_Type; Droite : Integer)
  return Date_Type;
- **function** "-" (Gauche : Date_Type; Droite : Integer)
  return Date_Type;

- \+ - * / ** **rem mod abs**
  = /= < > <= >=
  not and or xor &

La clause ‘Use type’

- Sans la clause

  Demain := Dates."+" (Ajourd_Hui, 1);

- Avec la clause

  use type Dates.Date_Type;

  Demain := Ajourd_Hui + 1;

Des pointeurs

- **type** Date_Access **is access** Date_Type;

- X : Date_Access;
- Y : Date_Access := **null**;
- Z : Date_Access := **new** Date_Type;

- X := **new** Date_Type;
- X := **new** Date_Type'(Jour => 25,Mois => Dec,An => 2006);
- X.**all**.Jour := 26;
- X.**all** := Y. **all**;
- X := Y;
- if X = **null** then ...

Une liste connectée

```
type Date_Type;

type Date_Access is access Date_Type;

type Date_Type is
   record
      Jour : Jour_Type;
      Mois : Mois_Type;
      An : AN_Type;
      Next : Date_Access;
   end record;
```

La suppression de la mémoire

- **procedure** Sup_Date **is new** Ada.Unchecked_Deallocation
  (Date_Type, Date_Access);

  X : Date_Access := **new** Date_Type;
  ...
  Sup_Date(X);

Des pointures générales (1)

- I : **aliased** Integer;
- **type** Array_Type **is array** (Positive **range** <>) **of aliased** Integer;
- **type** Record_Type **is**
  record
  I : **aliased** Integer;
  end record;

- **type** Integer_Access **is access all** Integer;

- IA : Integer_Access := I'Access;
  A : Array_Type(1..10);
  AA : Integer_Access := A(5)'Access;
  B : Record_Type;
  BA : Integer_Access := B.I'Access;

Des pointures générals (2)

```
procedure Illegal is
   type Integer_Access is access all Integer;
   IA : Integer_Access;
begin
   ...
   declare
      I : aliased Integer;
   begin
      IA := I'Access; -- l'assignation illégale
   end;
   IA.all := IA.all + 1; -- la variable "I" n'est pas existe.
end Illegal;
```

Des pointures générales (3)

```
procedure legal is
   type Integer_Access is access all Integer;
   IA : Integer_Access;
begin
   ...
   declare
      I : aliased Integer;
   begin
      IA := I'Unchecked_Access; -- dangereux
   end;
   IA.all := IA.all + 1; -- la variable “I” n’existe pas.
end legal;
```

Des pointures constants

- **type** Constant_Integer_Access **is access constant** Integer;

- Jours : **constant array** (1..7) **of** String (1..8) :=
  ("dimanche", "lundi   ", "mardi   ", "mercredi",
  "jeudi   ", "vendredi", "samedi ");

- Dim : **aliased constant** String := "dimanche";
  Lun : **aliased constant** String := "lundi";
  Mar : **aliased constant** String := "mardi";
  Mer : **aliased constant** String := "mercredi";
  Jeu : **aliased constant** String := "jeudi";
  Ven : **aliased constant** String := "vendredi";
  Sam : **aliased constant** String := "samedi";
  type Nom_Type **is access constant** String;
  Jours : **array** (1..7) **of** Nom_Type :=
  (Dim'Access, Lun'Access, Mar'Access, Mer'Access,
  Jeu'Access, Ven'Access, Sam'Access);

Des pointures constantes de procédures (1)

- **type** Menu_Operation **is access procedure**;
  procedure Add;
  procedure List;
  procedure Delete;
  Menu : **constant array** (1..3) **of** Menu_Operation :=
  (Add'Access, List'Access, Delete'Access);

- Menu(1).**all**;

Des pointures constantes de procédures (2)

- **type** Math_Func **is access function** (I : Float) **return** Float;
  function Sin (F : Float) **return** Float;
  function Cos (F : Float) **return** Float;
  function Tan (F : Float) **return** Float;
  Ops : **constant array** (1..3) **of** Math_Function :=
  (Sin'Access, Cos'Access, Tan'Access);

- F := Ops(1)(F); -- *= F := Ops(1).all(F);*

Des paramètres de pointure

- **function** F (A : **access** Integer) **return** Integer;
  procedure G (A : **access** Integer);

- **type** Integer_Access **is access** Integer;
  IA : Integer_Access := **new** Integer'(1);
  AI : **aliased** Integer;

- X : Integer := F(IA);
  Y : Integer := F(AI'Access);
  Z : Integer := F(**new** Integer);

Des discriminants de pointure

- ```
  type T (P : access Integer) is
     limited record
        ...
     end record;
  ```

- ```
  type Integer_Access is access Integer;
  Int_Access  : Integer_Access := new Integer'(1);
  Aliased_Int : aliased Integer;
  ```

- ```
  X : T (Int_Access);
  Y : T (Aliased_Int'Access);
  Z : T (new Integer);
  ```
  ```

Des paramètres généraux

- **generic**
  type Item_Type **is private**;
  package Lists **is**
  ...
  end Lists;

- **generic**
  type Item_Type(<>) **is private**;
  package Lists **is**
  ...
  end Lists;

- **type** Item_Type (Count : Integer) **is private**;

Ada.Unchecked_Deallocation

- **generic**
  type Object(<>) **is limited private**;
  type Name **is access** Object;
  procedure Ada.Unchecked_Deallocation (X : **in out** Name);

- **procedure** Sup_Date **is new** Ada.Unchecked_Deallocation
  (Date_Type, Date_Access);

Des sous-paquetages généraux (1)

- ```
  generic
     type Item_Type is private;
  package Lists is
     ...
  end Lists;
  ```
- ```
  generic
     type Other_Type is private;
  package Lists.Child is
     ...
  end Lists.Child;
  ```

- ```
  package Int_Lists is new Lists (Item_Type => Integer);
  ```
- ```
  package Int_List_Child is new Int_Lists.Child
                                (Other_Type => Boolean);
  ```

Des sous-paquetages généraux (2)

- ```
  generic
     type Item_Type is private;
  package Lists is
     ...
  end Lists;
  ```
- ```
  generic
  package Lists.Child is
     ...
  end Lists.Child;
  ```

- `package Int_Lists is new Lists (Item_Type => Integer);`
- `package Int_List_Child is new Int_Lists.Child;`

Toutes les possibilités de paramètre général

- limited private
- private
- (<>)
- range <>
- mod <>
- digits <>
- delta <>
- delta <> digits <>
- access Y
- access all Y
- access constant Y
- array (Y range <>) of Z
- array (Y) of Z
- new Y
- new Y with private
- abstract new Y with private
- tagged private
- tagged limited private
- abstract tagged private
- abstract tagged limited private

Des records variants

- ```
  type Une_Option is (C1, C2);
  type Un_Type (Choix : Une_Option) is
      record
          ...
          case Choix is
              when C1 => null;
              when C2 => R : Integer;
          end case;
      end record;
  ```

- ```
  A : Un_Type(C1);
  M : Un_Type(C2);
  ```

Des types étiquetés (1)

- ```
  type Un_Type is
     tagged record
        ...
     end record;
  ```

- ```
  type M_Type is new Un_Type with
     record
        R : Integer;
     end record;
  ```

- ```
  A : Un_Type;
  M : M_Type;
  A : Un_Type := Un_Type (M);
  M := (A with R=>101);
  ```
  ```

Des types étiquetés (2)

- **type** A_Type **is tagged private;**
- **type** B_Type **is new** A_Type **with private;**

Des opérations primaires (1)

- **package** TT **is**
  type A_Type **is tagged record** I : Integer; **end record**;
  procedure X (Appt : **in** A_Type); -- *primaire*

  type M_Type **is new** A_Type **with**
    record R : Integer; **end record**;
  function Y (B: **in** Integer) **return** A_Type; -- *primaire*
  end TT;

Des opérations primaires (2)

- **package** TT **is**
  type A_Type **is tagged record** I : Integer; **end record**;
  procedure X (Appt : **in** A_Type); -- *primaire*

  type M_Type **is new** A_Type **with**
    record R : Integer; **end record**;
  procedure Z (Appt : **in** A_Type); -- *La déclaration Illégale*
  function Y (B: **in** Integer) **return** A_Type; -- *primaire*
  end TT;

Les opérations primaires (3)

```
package TT is
   type A_Type is tagged record I : Integer; end record;
   procedure X (Appt : in A_Type);
   function Y (B: in Integer) return A_Type;

   type M_Type is new A_Type with
      record R : Integer; end record;

   function Y (B: in Integer) return M_Type;
end TT;
```

Des types de domaine de classe(1)

- **type** A_Type **is tagged record** ... **end record**;
- **type** B_Type **is new** A_Type **with record** ... **end record**;
- **type** C_Type **is new** A_Type **with record** ... **end record**;
- **type** A_Access **is access** A_Type'Class;
- B : A_Access := **new** B_Type;
- C : A_Access := **new** C_Type;

Des types de domaine de classe(2)

- ```
  procedure X (Y : in A_Type'Class) is
  begin
     if Y in B_Type'Class then
        Put_B(Y); -- Y est une variable de B_Type
     elsif Y in C_Type'Class then
        Put_C(Y);-- Y est une variable de C_Type
     else
        Put_A(Y);-- Y est une variable de A_Type
     end if;
  end X;
  ```

- ```
  if Y'Tag = A_Type'Tag then ...
  ```

Le dispatching automatique (1)

- **procedure** X (Y : **in** A'Class) **is**
  begin
  Put(Y); -- *si “Put” est une procédure primaire de A_Type*
  end X;

Le dispatching automatique (2)

- **type** A_Type **is tagged record** ... **end record**;
- **type** O_Type **is tagged record** ... **end record**;
- -- *La déclaration illégale*
  procedure Put (A : **in** A_Type; O : **in** O_Type);
- -- *La déclaration légale*
  procedure Put (A, B : **in** A_Type);
  procedure Put (A : **in** A_Type; O : **in** O_Type'Class);

Des types abstraits

- **type** Shape **is tagged record null**; **end record**;
- **type** Shape **is tagged null record**;
- **type** A_Shape **is new** Shape **with null record**;
- **type** Shape **is abstract tagged null record**;
- **procedure** Draw (S : **in** Shape) **is abstract**;
- **type** C_Shape **is abstract new** Shape **with**
  record ... **end record**;

Flux entrée/sortie (1)

- **procedure** A_Type'Class'Output (
  Stream : **access** Ada.Streams.Root_Stream_Type'Class;
  Item : **in** A_Type'Class);

- **function** A_Type'Class'Input (
  Stream : **access** Ada.Streams.Root_Stream_Type'Class)
  return A_Type'Class;

Flux entrée/sortie (2)

```
procedure Save (A : in A_Type; To : in String) is
   File   : Ada.Streams.Stream_IO.File_Type;
   Stream : Ada.Streams.Stream_IO.Stream_Access;
begin
   Ada.Streams.Stream_IO.Create (File, Name => To);
   Stream := Ada.Streams.Stream_IO.Stream(File);
   A_Type'Class'Output (Stream, A);
   Ada.Streams.Stream_IO.Close (File);
end Save;
```

Flux entrée/sortie (3)

```
procedure Load (A : in out A_Type; From  : in String) is
   File   : Ada.Streams.Stream_IO.File_Type;
   Stream : Ada.Streams.Stream_IO.Stream_Access;
begin
   Ada.Streams.Stream_IO.Open (File, Name => From,
                               Mode => Ada.Streams.Stream_IO.In_File);
   Stream := Ada.Streams.Stream_IO.Stream(File);
   A := A_Type(A_Type'Class'Input (Stream));
   Ada.Streams.Stream_IO.Close (File);
exception
   when Ada.Streams.Stream_IO.Name_Error =>
      raise A_Error;
end Load;
```

Les types gouvernés (1)

- **procedure** Finalize (Object : **in out** Controlled);
  procedure Finalize (Object : **in out** Limited_Controlled);

- **type** List_Type **is new** Limited_Controlled **with**
  record
  ...
  end record;
  procedure Finalize (Object : **in out** List_Type);

- **procedure** Initialize (Object : **in out** Controlled);
  procedure Initialize (Object : **in out** Limited_Controlled);

Les types gouvernés (2)

- A := B; -- compiled as: Temp := B;
  -- Finalize(A);
  -- A := B;
  -- Adjust(A);
  -- Finalize(Temp);

- A := B; -- compiled as: Finalize(A);
  -- A := B;
  -- Adjust(A);

- **procedure** Adjust (Object : **in out** Controlled);