Le langage Ada

Raphael Amiard - Adacore
amiard@adacore.com

Historique

- Dans les années 70, le DOD souffre d'une explosion du nombre des langages utilisés
- Il lance un concours international pour un langage qui répond à toutes ces exigences (1974)

Historique

- Plusieurs propositions
- Vainqueur: L'équipe de Jean Ichbiah, CII Honeywell Bull
- Première normalisation du langage (ANSI, ISO): 1983
- Révisions majeures en 1995, 2005, 2012.
- L'un des seuls langages normalisés à priori

Exigences du langage

- Généraliste
  - Efficacité
  - Simplicité
  - Implémentabilité
- Haut niveau de génie logiciel
  - Maintenabilité
  - Portabilité
  - Fiabilité
- Norme claire et non ambiguë
- Travail sur des plateformes embarquées
- Traitements parallèles
- Gestion des données bas niveau

Résultat: Ada

- Dérivé d'une syntaxe type Pascal
- Impératif (comme Fortran, Cobol, C/C++, Java, Python...)
- Parallélisme intégré au langage (par opposition aux API type pthread)
- Modulaire (facilité de mise en place de sous-ensembles)
- Vérifications statiques et dynamiques (bornes...)

Ada aujourd'hui

Marché privilégié:

- Systèmes temps-réel
- Systèmes critiques (safety critical, mission critical)
- Systèmes de sécurité (MILS)

Exemples

- Arianne 6
- 787 Dreamliner (Common Core System)
- Airbus A350 XWB (Air Data Inertial Reference Unit)
- Sentinel 1 (Environmental Satellite System)
- Canadian Space Arm
- Meteor (metro line 14)

Exemple: Hello, World

```
with Ada.Text_IO; use Ada.Text_IO;

--  Display a welcome message
procedure Greet is
begin
    Put_Line ("Hello, world!");
end Greet;
```

```
$ gnatmake greet.adb
$ ./greet
```

Imperative language

Imperative language - for loops

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
begin
   for I in 1 .. 10 loop
      Put_Line("Hello, World!");
   end loop;
end Greet;
```

- I here denotes a constant that is only accessible in the loop.
- “1 .. 10” is a range.
- Put_Line is a procedure call. procedure is like a fn returning void in C/C++.

Imperative language - while loops

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
   --  Variable declaration. Only legal in declarative
   --  parts.
   I : Integer := 1;
begin
   --  Condition. *Must* be of type boolean
   while I < 10 loop
      Put_Line("Hello, World!");

      --  Assignment
      I := I + 1;
   end loop;
end Greet;
```

Imperative language - General loops

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
   I : Integer := 1;
begin
   loop
      Put_Line("Hello, World!");
      exit when I = 5;
      --  Exit statement - takes a boolean condition
      I := I + 1;
   end loop;
end Greet;
```

Imperative language - If

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
   I : Integer := 1;
begin
   loop
      Put_Line("Hello, World!");
      if I = 5 then
         exit;
      end if;
      I := I + 1;
   end loop;
end Greet;
```

Imperative language - If/Else

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
   I : Integer := 1;
begin
   loop
      if I = 5 then
         exit;
      else
         Put_Line("Hello, World!");
      end if;
      I := I + 1;
   end loop;
end Greet;
```

Imperative language - If/Elsif/Else

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
   I : Integer := 0;
begin
   loop
      if I = 5 then
         exit;
      elsif I = 0 then
         Put_Line ("Starting...");
      else
         Put_Line ("Hello, World!");
      end if;
      I := I + 1;
   end loop;
end Greet;
```

Imperative language - If/Else

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
   I : Integer := 1;
begin
   loop
      --  "or else" is the short circuit or operator
      if I = 5 or else I = 2 then
         exit;
      else
         Put_Line("Hello, World!");
      end if;
      I := I + 1;
   end loop;
end Greet;
```

Imperative language - If/Else

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
   I : Integer := 1;
begin
   loop
      --  "and then" is the short circuit or operator
      if I < 5 and then I > 2 then
         exit;
      else
         Put_Line("Hello, World!");
      end if;
      I := I + 1;
   end loop;
end Greet;
```

Imperative language - Case statement

```
procedure Greet is
   I : Integer := 0;
begin
   loop
      -- Expression must be of a discrete type. All the
      -- values must be covered.
      case I is
         when 0 => Put_Line ("Starting...");
         when 3 .. 4 => Put_Line ("Hello");
         when 7 | 9 => exit;
         -- 'when others' must be the last one and alone (if
         -- present)
         when others => Put_Line ("Hello, World!");
      end case;
      I := I + 1;
   end loop;
end Greet;
```

Quizz: Imperative language

Quizz 1: Is there a compilation error?

```
for I in 10 .. 1 loop
    Put_Line("Hello, World!");
end loop;
```

Quizz 2: Is there a compilation error?

```
for I in reverse 1 .. 10 loop
    Put_Line("Hello, World!");
end loop;
```

Quizz 3: Is there a compilation error?

```
procedure Hello is
   I : Integer;
begin
   for I in 1 .. 10 loop
      Put_Line ("Hello, World!");
   end loop;
end Hello;
```

Quizz 4: Is there a compilation error?

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   I : Integer;
begin
   while I < 10 loop
      Put_Line("Hello, World!");
      I := I + 1;
   end loop;
end Greet;
```

Quizz 5: Is there a compilation error?

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
    I : Integer := 2;
begin
    while i < 10 loop
        Put_Line ("Hello, World!");
        i := i + 1;
    end loop;
end Greet;
```

Quizz 6: Is there a compilation error?

```
with Ada.Text_IO; use Ada.Text_IO;
with Tools;

procedure Greet is
begin
    loop
        Put_Line("Hello, World!");
        Tools.My_Proc;
    end loop;
end Greet;
```

Quizz 7: Is there a compilation error?

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
   I : Integer := 0;
begin
   loop
      if I = 5 then
         exit;
      else
         if I = 0 then
            Put_Line ("Starting...");
         else
            Put_Line ("Hello, World!");
         end if;
      end if;
      I := I + 1;
   end loop;
end Greet;
```

Quizz 8: Is there a compilation error?

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
   I : Integer := 0;
begin
   loop
      case I is
         when 0 =>
            Put_Line ("Starting...");
         when 1 .. 4 =>
            Put_Line ("Hello");
         when 5 =>
            exit;
      end case;
      I := I + 1;
   end loop;
end Greet;
```

Quizz 9: Is there a compilation error?

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
begin
   loop
      case I is
         when 0 =>
            Put_Line ("Starting...");
         when 1 .. 4 =>
            Put_Line ("Hello");
         when others =>
            exit;
      end case;
      I := I + 1;
   end loop;
end Greet;
```

Quizz 10: Is there a compilation error?

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
   I : Integer := 0;
begin
   loop
      case I is
         when Integer'First .. 1 =>
            Put_Line ("Starting...");
         when 1 .. 4 =>
            Put_Line ("Hello");
         when others =>
            exit;
      end case;
      I := I + 1;
   end loop;
end Greet;
```

Quizz 11: Which one is an error?

```
V    : Integer;
1V   : Integer;
V_   : Integer;
_V   : Integer;
V__1 : Integer;
V_1  : Integer;
```

Strongly typed language

Question

What is a type?

Integers

- Integer types are just regular types (not a built-in)

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   type My_Int is range 1 .. 20;
   --  Declare a signed integer type, and give the bounds

   --  Like variables, declarations can only happen in
   --  declarative region
begin
   for I in My_Int loop
      Put_Line("Hello, World!");
   end loop;
end Greet;
```

Integers

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   type My_Int is range 1 .. 20;
begin
   --  Iterates on every value in the range of My_Int
   for I in My_Int loop
      if I = My_Int'Last then
      --           ^ Higher bound of the type
      --  'First is the lower bound
         Put_Line ("Bye");
      else
         Put_Line("Hello, World!");
      end if;
   end loop;
end Greet;
```

Integers

```
procedure Greet is
   A : Integer := Integer'Last;
   B : Integer;
begin
   B := A + 5;
   --  This operation will overflow, eg. it will
   --  raise an exception at runtime.
end Greet;
```

Integer

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   type My_Int is range 1 .. 20;
   A : My_Int := 12;
   B : My_Int := 15;
   M : My_Int := (A + B) / 2;
   --  No overflow here, overflow checks are done at
   --  specific boundaries.
begin
   for I in 1 .. M loop
      Put_Line("Hello, World!");
   end loop;
end Greet;
```

Enumerations

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday, Saturday, Sunday);
   --  An enumeration type
begin
   for I in Days loop
      case I is
         when Saturday .. Sunday =>
            Put_Line ("Week end!");

         --  Completeness checking on enums
         when others =>
            Put_Line ("Hello on " & Days'Image (I));
            --  'Image attribute, converts a value to a
            --  String
      end case;
   end loop;
end Greet;
```

Strong typing

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   --  Declare two signed types
   type Meters is range 0 .. 10_000;
   type Miles is range 0 .. 5_000;

   Dist_Us : Miles;
   --  Declare a constant
   Dist_Eu : constant Meters := 100;
begin
   --  Not correct: types mismatch
   Dist_Us := Dist_Eu * 1609 / 1000;
   Put_Line (Miles'Image (Dist_Us));
end Greet;
```

Strong typing

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Conv is
   type Meters is range 0 .. 10_000;
   type Miles is range 0 .. 5_000;
   Dist_Us : Miles;
   Dist_Eu : constant Meters := 100;
begin
   Dist_Us := Miles (Dist_Eu * 1609 / 1000);
   --          ^ Type conversion, from Meters to Miles
   --  Now the code is correct

   Put_Line (Miles'Image (Dist_Us));
end;
```

Strong typing

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   C : Character;
   --  ^ Built-in character type (it's an enum)
begin
   C := '?';
   --   ^ Character literal (enumeration literal)

   C := 64;
   --   ^ Invalid: 64 is not an enumeration literal
end Greet;
```

Strong typing

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
   C : Character;
begin
   C := '?';
   Put_Line ("""Ascii"" code of '" & C & "' is"
   --         ^ Use "" to insert quote in a string
             & Integer'Image (Character'Pos (C)));
   --                                ^ 'Pos converts a
   --                                   value to its position

   C := Character'Val (64);
   --            ^ 'Val converts a position to its value
end Greet;
```

Subtypes

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
   type Days is (Monday, Tuesday, Wednesday, Thursday,
                 Friday, Saturday, Sunday);

   --  Declaration of a subtype
   subtype Weekend_Days is Days range Saturday .. Sunday;
   --                              ^ Constraint of the subtype
begin
   for I in Days loop
      case I is
         --  Just like a type, a subtype can be used as a
         --  range
         when Weekend_Days =>
            Put_Line ("Week end!");
         when others =>
            Put_Line ("Hello on " & Days'Image (I));
      end case;
   end loop;
end Greet;
```

A subtype doesn't define a type

- All subtypes are of the same type.

```
with Ada.Text_IO; use Ada.Text_IO;

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

   subtype Weekend_Days is Days range Saturday .. Sunday;
   Day : Days := Saturday;
   Weekend : Weekend_Days;
begin
   Weekend := Day;
   --         ^ Correct: Same type
   Weekend := Monday;
   --         ^ Wrong value for the subtype
   --  Compiles, but exception at runtime
end Greet;
```

Quizz: Types

Quizz 1: Is there a compilation error?

```
type My_Int is range 1 .. 20.5;
```

Quizz 2: Is there a compilation error?

```
type My_Int is range 1 .. 20.0;
```

Quizz 3: Is there a compilation error?

```
A : Integer := 5;
type My_Int is range A .. 20;
```

Quizz 4: Is there a compilation error?

```
type My_Int is range 1 .. Integer'Last;
```

Quizz 5: Is there a compilation error?

```
type My_Int_1 is range 1 .. Integer'Last;
type My_Int_2 is range Integer'First .. 0;
type My_Int_3 is range My_Int_2'First .. My_Int_2'Last;
```

Quizz 6: Is there a compilation error?

```
type My_Int_1 is range 1 .. Integer'Last;
subtype My_Int_2 is My_Int_1 range 1 .. 100;

V1 : My_Int_1 := 5;
V2 : My_Int_2;
V2 := V1;
```

Quizz 7: Is there a compilation error?

```
type My_Int_1 is range 1 .. Integer'Last;
type My_Int_2 is range 1 .. 100;

V1 : My_Int_1 := 5;
V2 : My_Int_2;
V2 := V1;
```

Quizz 8: Is there a compilation error?

```
type Enum is (E1, E2);
type Enum2 is (E2, E3);
```

Quizz 9: Is there a compilation error?

```
type Bit is ('0', '1');
```

Arrays

Array type declaration

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   type My_Int is range 0 .. 1000;
   type Index is range 1 .. 5;

   type My_Int_Array is array (Index) of My_Int;
   --                                    ^ Type of elements
   --                           ^ Bounds of the array
   Arr : My_Int_Array := (2, 3, 5, 7, 11);
begin
   for I in Index loop
      Put (My_Int'Image (Arr (I)));
   end loop;
   New_Line;
end Greet;
```

Array type declaration

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   type My_Int is range 0 .. 1000;
   type Index is range 1 .. 5;
   type My_Int_Array is array (Index) of My_Int;

   Arr : My_Int_Array := (2, 3, 5, 7, 11);
   --                    ^ Aggregate (array literal)
begin
   for I in Index loop
      Put (My_Int'Image (Arr (I)));
   end loop;
   New_Line;
end Greet;
```

Array index

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   type My_Int is range 0 .. 1000;
   type Index is range 11 .. 15;
   --                  ^ Low bound can be any value
   type My_Int_Array is array (Index) of My_Int;
   Tab : My_Int_Array := (2, 3, 5, 7, 11);
begin
   for I in Index loop
      Put (My_Int'Image (Tab (I)));
   end loop;
   New_Line;
end Greet;
```

Array index

```
procedure Greet is
   type My_Int is range 1 .. 31;
   type Month is (Jan, Feb, Mar, Apr, May, Jun,
                  Jul, Aug, Sep, Oct, Nov, Dec);

   type My_Int_Array is array (Month) of My_Int;
   --                           ^ Can use an enum as the
   --                             index

   Tab : constant My_Int_Array :=
   --    ^ constant is like a variable but cannot be
   --      modified
     (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
   --  Maps months to number of days

   Feb_Days : My_Int := Tab (Feb);
   --  Number of days in February
begin
   for I in Month loop
      Put_Line (My_Int'Image (Tab (I)));
   end loop;
end Greet;
```

Indexation

```
procedure Greet is
   type My_Int is range 0 .. 1000;
   type Index is range 1 .. 5;
   type My_Int_Array is array (Index) of My_Int;
   Tab : My_Int_Array := (2, 3, 5, 7, 11);
begin
   Indexation
   for I in 2 .. 6 loop
      Put (My_Int'Image (Tab (I)));
      --                      ^ Indexation
      --                      ^ Will raise an exception when
      --                      I = 6
   end loop;
   New_Line;
end Greet;
```

Indexation

```
procedure Greet is
   type My_Int is range 0 .. 1000;
   type Index is range 1 .. 5;
   type My_Int_Array is array (Index) of My_Int;
   Tab : My_Int_Array := (2, 3, 5, 7, 11);
begin
   for I in Integer range 1 .. 5 loop
   --       ^ We say that the type of I is Integer
      Put (My_Int'Image (Tab (I)));
   --                           ^ Compile time error
   end loop;
   New_Line;
end Greet;
```

Shortcut for index

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   type My_Int is range 0 .. 1000;
   type My_Int_Array is array (1 .. 5) of My_Int;
   --                           ^ Subtype of Integer
   Tab : My_Int_Array := (2, 3, 5, 7, 11);
begin
   for I in 1 .. 5 loop
   --       ^ Likewise
      Put (My_Int'Image (Tab (I)));
   end loop;
   New_Line;
end Greet;
```

Range attribute

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   type My_Int is range 0 .. 1000;
   type My_Int_Array is array (1 .. 5) of My_Int;
   Tab : My_Int_Array := (2, 3, 5, 7, 11);
begin
   for I in Tab'Range loop
   --          ^ Gets the range of Tab
      Put (My_Int'Image (Tab (I)));
   end loop;
   New_Line;
end Greet;
```

Unconstrained arrays

```
procedure Greet is
   type Days is (Monday, Tuesday, Wednesday,
      Thursday, Friday, Saturday, Sunday);

   type Workload_Type is array (Days range <>) of Natural;
   --  Indefinite array type
   --                            ^ Bounds are of type Days,
   --                              but not known

   Workload : constant Workload_Type (Monday .. Friday) :=
   --                                  ^ Specify the bounds
   --                                    when declaring
      (Friday => 7, others => 8);
   --                ^ Default value
   --  ^ Specify element by name of index
begin
   for I in Workload'Range loop
      Put_Line (Integer'Image (Workload (I)));
   end loop;
end Greet;
```

Declaring arrays

```
with Ada.Text_IO; use Ada.Text_IO;

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

   type Workload_Type is array (Days range <>) of Natural;

   Workload : constant Workload_Type :=
      (Monday .. Thursday => 8, Friday => 7);
   --  ^ More powerful association by name
   --  Here, no need to specify the bounds of the array
begin
   for I in Workload'Range loop
      Put_Line (Integer'Image (Workload (I)));
   end loop;
end Greet;
```

Predefined array type: String

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   Message : String (1 .. 11) := "dlroW olleH";
   --        ^ Pre-defined array type.
   --          Component type is Character
begin
   for I in reverse 1 .. 11 loop
            ^ Iterate in reverse order
      Put (Message (I));
   end loop;
   New_Line;
end Greet;
```

Predefined array type: String

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   Message : constant String := "Hello World";
   --                ^ Bounds are automatically computed
   --                  from initialization value
begin
   for I in reverse Message'First .. Message'Last loop
   --                      ^ 'First and 'Last return the low and
   --                      high bound
   -- (But you should use 'Range most of the time)
      Put (Message (I));
   end loop;
   New_Line;
end Greet;
```

Declaring arrays

```
with Ada.Text_IO; use Ada.Text_IO;

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

   subtype Day_Name is String (1 .. 2);
   --  Subtype of string with known size

   type Days_Name_Type
   is array (Days) of Day_Name;
   --        ^ Type of the index
   --                  ^ Type of the element. Must be
   --                    definite
begin
   null;
end Greet;
```

Declaring arrays

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday, Saturday, Sunday);
   subtype Day_Name is String (1 .. 2);
   type Days_Name_Type is array (Days) of Day_Name;
   Names : constant Days_Name_Type :=
     ("Mo", "Tu", "We", "Th", "Fr", "Sa", "Su");
   --  Initial value given by aggregate
begin
   for I in Names'Range loop
      Put_Line (Names (I));
   end loop;
end Greet;
```

Declaring arrays

```
--  Those two declarations produce the same thing
A : String := "Hello World";
B : String (1 .. 11) := ('H', 'e', 'l', 'l', 'o', ' ',
                         'W', 'o', 'r', 'l', 'd');
```

Quizz: Arrays

Quizz 1: Is there a compilation error ?

```
--  Natural is a pre-defined subtype.
subtype Natural is Integer range 0 ..  Integer'Last

type Arr is array (Natural) of Integer;
Name : Arr;
```

Quizz 2: Is there a compilation error ?

```
type Arr is array (Natural range <>) of Integer;
Name : Arr;
```

Quizz 3: Is there a compilation error ?

```
type Str_Array is array (1 .. 10) of String;
```

Quizz 4: Is there a compilation error ?

```
A : constant Integer := 5;
```

Quizz 5: Is there a compilation error ?

```
A : constant String (1 .. 12);
```

Quizz 6: Is there a compilation error ?

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday, Saturday, Sunday);
   type WorkLoad_Type is array (Days range <>) of Natural;
   Workload : constant Workload_Type :=
      (Monday .. Friday => 8, Friday => 7, Saturday | Sunday => 0);
begin
   for I in Workload'Range loop
      Put_Line (Integer'Image (Workload (I)));
   end loop;
end Greet;
```

Modular/Structured programming

Packages

```
package Week is
end Week;
```

Packages:

- Group related declarations together
- Define an interface (API)
- Hide the implementation
- Provide a name space

Packages

```
package Week is

   --  This is a declarative part. You can put only
   --  declarations here, no statements

   type Days is (Monday, Tuesday, Wednesday,
      Thursday, Friday, Saturday, Sunday);
   type WorkLoad_Type is array (Days range <>) of Natural;
   Workload : constant Workload_Type :=
      (Monday .. Friday => 8, Friday => 7, Saturday | Sunday => 0);
end Week;
```

Different from header files in C/C++ because:

- Language level mechanism (not a preprocessor)
- Not text based
- With'ing a package does not "copy/paste" the content of the spec into your file
- With GNAT, packages specs go in .ads files (here, it would be week.ads)

With-ing a package

```
with Ada.Text_IO; use Ada.Text_IO;

with Week;
--  references the package, and adds a dependency on
--  that package

procedure Greet is
begin
   for I in Week.Workload'Range loop
   --       ^ We reference items of the package by prefixing
   --         by package name
      Put_Line (Integer'Image (Workload (I)));
   end loop;
end Greet;
```

Using a package

```
with Ada.Text_IO; use Ada.Text_IO;

with Week;
use Week;
--  Brings the content of the package in the current
--  namespace

procedure Greet is
begin
   for I in Workload'Range loop
   --       ^ We can reference items of the package directly
   --         now
      Put_Line (Integer'Image (Workload (I)));
   end loop;
end Greet;
```

Package body

```
package body Week is

   --  The body contains additional declarations, not
   --  visible from the spec, or anywhere outside of the
   --  body
   type WorkLoad_Type is array (Days range <>) of Natural;
   Workload : constant Workload_Type :=
      (Monday .. Friday => 8, Friday => 7, Saturday | Sunday => 0);

   function Get_Workload (Day : Days) return Natural is
   begin
      return Workload (Day);
   end;
   --  This is a function. It has a return statement
   --  (mandatory for functions)
end Week;
```

- With GNAT, packages bodies go in .adb files (here, it would be week.adb)

Subprograms

Subprograms

```
with Ada.Text_IO; use Ada.Text_IO;

--  Here we declare and define a procedure without
--  parameters
procedure Greet is
begin
   Put_Line("Hello, World!");
end Greet;
```

Subprograms

```
package Week is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday, Saturday, Sunday);

   function Get_Workload (Day : Days) return Natural;
   --  We declare (but don't define) a function with one
   --  parameter, returning a Natural integer
end Week;
```

Subprograms

```
package Week is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday, Saturday, Sunday);

   function Get_Day_Name
      (Day : Days := Monday) return String;
   --                              ^ We can return any type,
   --                                even indefinite ones
   --            ^ Default value for parameter
end Week;
```

Subprograms

```
package body Week is
   --  Implementation of the Get_Day_Name function
   function Get_Day_Name (Day : Days := Monday) return String is
   begin
      case Day is
         when Monday => return “Monday”;
         when Tuesday => return “Tuesday”;
         ...
         when Sunday => return “Sunday”;
      end case;
   end Get_Day_Name;
end Week;
```

Parameters modes

```
package Week is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday, Saturday, Sunday);
   procedure Next_Day (Res : out Days;
   --                          ^ Mode out: not initialized at
   --                            the beginning, procedure body can
   --                            assign it. (Like a return value)
                       Day : Days);
   --                          ^ No mode: defaults to "in",
   --                            which is like a constant
end Week;
```

Parameters modes

```
--                You can declare several params at the same
--              v time
procedure Swap (A, B : in out Integer)
--                     ^ In out is initialized at the
--                       beginning with value passed by
--                       caller. procedure can assign it,
--                       caller will get new value. Like
--                       pass by reference
--                       Use it instead of a pointer in C
is
   Tmp : Integer;
begin
   Tmp := A;
   A := B;
   B := Tmp;
   return;
   --  Return statement. Optional for procedures
end Swap;
```

Subprogram call

```
procedure Test_Swap
is
   X, Y : Integer;
begin
   X := 5;
   Y := 7;
   Swap (X, Y);
   --    ^ Positional parameters
   Swap (A => X, B => Y);
   --    ^ Named parameters
   Swap (B => X, A => Y);
   --    ^ You can reverse the order
end Test_Swap;
```

Function calls

```
function Double (I : Integer) return Integer is
begin
   return I * 2;
end Double;

function Quadruple (I : Integer) return Integer is
   Res : Integer := Double (Double (I));
   --               ^ Calling the double function
begin
   Double (I);
   --  ILLEGAL: You cannot ignore a function's return value

   return Res;
end Quadruple;
```

Function calls

```
function Pi return Float is
--          ^ No parameters, no parentheses
begin
   return 3.1419;
end Pi;

A : Float := Pi;
--           ^ No parameters, no parentheses at call site
--             either!
```

Mutually recursive subprograms

```
procedure Compute_A (V : Natural);
--  Forward declaration of Compute_A

procedure Compute_B (V : Natural) is
begin
   if V > 5 then
      Compute_A (V - 1);
   -- ^ Call to Compute_A
   end if;
end Compute_B;

procedure Compute_A (V : Natural) is
begin
   if V > 2 then
      Compute_B (V - 1);
   -- ^ Call to Compute_B
   end if;
end Compute_A;
```

Nested subprograms

```
function Quadruple (I : Integer) return Integer is

   function Double (I : Integer) return Integer is
   begin
      return I * 2;
   end Double;
   --  Nested function

begin
   return Double (Double (I));
end Quadruple;
```

Quizz: Packages & subprograms

Quizz 1: Is there a compilation error ?

```
package My_Type is
   type My_Type is range 1 .. 100;
end My_Type;
```

Quizz 2: Is there a compilation error ?

```
package Pkg is
   function F (A : Integer);
end Pkg;
```

Quizz 3: Is there a compilation error ?

```
package Pkg is
   function F (A : Integer) return Integer;
   function F (A : Character) return Integer;
end Pkg;
```

Quizz 4: Is there a compilation error ?

```
package Pkg is
   function F (A : Integer) return Integer;
   procedure F (A : Character);
end Pkg;
```

Quizz 5: Is there a compilation error ?

```
package Pkg is
   subtype Int is Integer;
   function F (A : Integer) return Integer;
   function F (A : Int) return Integer;
end Pkg;
```

Quizz 6: Is there a compilation error ?

```
package Pkg is
   procedure Proc (A : Integer);
   procedure Proc (A : in out Integer);
end Pkg;
```

Quizz 7: Is there a compilation error ?

```
package Pkg is
   procedure Proc (A : in out Integer := 7);
end Pkg;
```

Quizz 8: Is there a compilation error ?

```
package Pkg is
   procedure Proc (A : Integer := 7);
end Pkg;

package body Pkg is
   procedure Proc (A : Integer) is
      ...
   end Proc;
end Pkg;
```

Quizz 9: Is there a compilation error ?

```
package Pkg is
   procedure Proc (A : in out Integer);
end Pkg;

package body Pkg is
   procedure Proc (A : in out Integer) is
      ...
   end Proc;

   procedure Proc (A : in out Character) is
      ...
   end Proc;
end Pkg;
```

Quizz 10: Is there a compilation error ?

```
package Pkg is
   procedure Proc (A : in Integer);
end Pkg;

package body Pkg is
   procedure Proc (A : in Integer);
   procedure Proc (A : in Integer) is
      ...
   end Proc;
end Pkg;
```

Quizz 11: Is there a compilation error ?

```
-- pkg1.ads
package Pkg1 is
    procedure Proc;
end Pkg1;

-- pkg2.ads
with Pkg1;
package Pkg2 is
...
end Pkg2;

-- main.adb
with Pkg2;

procedure Main is
begin
   Pkg1.Proc
end Main;
```

Quizz 12: Is there a compilation error ?

```
--  pkg1.ads
package Pkg1 is
  procedure Proc;
  ...
end Pkg1;

--  pkg2.ads
with Pkg1; use Pkg1;
package Pkg2 is
   ...
end Pkg2;

--  pkg2.adb
package body Pkg2 is
   procedure Foo is
   begin
      Proc;
   end Foo;
end Pkg2;
```

Quizz 13: Is there a compilation error ?

```
--  pkg1.ads
package Pkg1 is
procedure Proc;
   ...
end Pkg1;

--  pkg2.ads
with Pkg1; use Pkg1;
package Pkg2 is
   ...
end Pkg2;

--  pkg2.adb
with Pkg1; use Pkg1;
package body Pkg2 is
   ...
end Pkg2;
```

Quizz 14: Is there a compilation error ?

```
--  pkg1.ads
package Pkg1 is
   procedure Proc;
   ...
end Pkg1;

--  pkg2.ads
with Pkg1;
package Pkg2 is
   ...
end Pkg2;

--  pkg2.adb
use Pkg1;

package body Pkg2 is
   procedure Foo is
   begin
      Proc;
   end Foo;
end Pkg2;
```

More about types

Array

```
Len : Natural := F (5);

--  The size of this array is not known at compile time. But the bounds are
--  fixed.
Buf : String (1 .. Len);

--  This does not change the size of the array.
Len := 3;
```

Array slices

```
procedure Main is
    Buf : String := "Hello ...";

    Full_Name : "Raphael Amiard";
begin
    Buf (7 .. 9) := "Bob";
    Put_Line (Buf);  --  Prints "Hello Bob"

    Put_Line ("Hi " & Full_Name (1 .. 7)); --  Prints "Hi Raphael"
end;
```

Records

```
type Date is record
   --  The following declarations are components of the record
   Day   : Integer range 1 .. 31;
   Month : Month_Name;
   Year  : Integer range 1 .. 3000; --  You can add custom constraints on fields
end record;
```

Records - default values

```
type Date is record
   Day   : Integer range 1 .. 31;
   Month : Month_Name := January;

   --  This component has a default value
   Year  : Integer range 1 .. 3000 := 2012;
   --                                 ^ Default value
end record;
```

Records - Literals

```
Today    : Date := (31, November, 2012);
Birthday : Date := (Day => 30, Month => February, Year => 2010);
--                  ^ By name
```

Records - Selection

```
Today.Day := 29;
```

Access types (pointers)

```
--  Declare an access type
type Date_Acc is access Date;

D : Date_Acc;
D := null;
--   ^ Literal for "access to nothing"
```

Dereferencing

```
type Date_Acc is access Date;
D : Date_Acc;

Today : Date := D.all;  --  Access derefence
J     : Day := D.Day
--              ^ Implicit dereference for record and array components
--                Equivalent to D.all.day
```

Allocation (by type)

```
D : Date_Acc := new Date;  --  Allocation (using default values)
```

Allocation (by type)

```
type String_Acc is access String;
--                          ^ Access to unconstrained array type
Msg : String_Acc;
--    ^ Default value is null

Buffer : String_Acc := new String (1 .. 10);
--                                 ^ Constraint required
```

Allocation (by expression)

```
D   : Date_Acc := new Date'(30, November, 2011);
Msg : String_Acc := new String'("Hello");
```

Mutually recursive types

```
type Node;
--  This is an incomplete type declaration, it must be
--  completed in the same declarative region.

type Node_Acc is access Node;

type Node is record
   Content    : Natural;
   Prev, Next : Node_Acc;
end record;
```

More about records

```
Max_Len : constant Natural := Compute_Max_Len;
--                            ^ Not known at compile time

type Person is record
   First_Name : String (1 .. Max_Len);
   Last_Name  : String (1 .. Max_Len);
end record;
```

Records with discriminant

```
type Person (Max_Len : Natural) is record
--           ^ Discriminant. Cannot be modified once initialized.
   First_Name : String (1 .. Max_Len);
   Last_Name  : String (1 .. Max_Len);
end record;
--  Person is an indefinite type (like an array)
```

Records with variant

```
type Node_Acc is access Node;

type Op_Kind is (Bin_Op, Un_Op);
--  A regular enum

type Node (Op : Op_Kind) is record
--         ^ The discriminant is an enum
   Id : Natural;
   case Op is
      when Un_Op =>
         Operand : Node_Acc;
      when Bin_Op =>
         Left, Right : Node_Acc;
      --  Those fields only exist when Op is Bin_Op
   end case;
   --  Variant part. Only one, at the end of the record
   --  definition
end record;
```

Quizz: More about types

Quizz 1: Is there a compilation error?

```
Buf : String (1 .. 10);
...
Buf (2 .. 4) := "Ab";
```

Quizz 2: Is there a compilation error?

```
type Person (Max_Len : Natural) is record
   First_Name : String (1 .. Max_Len);
   Last_Name  : String (1 .. Max_Len);
end record;

A : Person;
```

Quizz 3: Is there a compilation error?

```
type Person (Max_Len : Natural) is record
   Name : String (1 .. Max_Len);
end record;

A : Person (20);
```

Quizz 4: Is there a compilation error?

```
type Person (Max_Len : Natural) is record
   Name : String (1 .. Max_Len);
end record;

A : Person := Person'(6, "Pierre");
```

Quizz 5: Is there a compilation error?

```
type Person (Max_Len : Natural) is record
   Name : String (1 .. Max_Len);
end record;

A : Person := Person'(20, "Pierre");
```

Quizz 6: Is there a compilation error?

```
type Date1_Acc is access Date;
type Date2_Acc is access Date;

D1 : Date1_Acc;
D2 : Date2_Acc;

D1 := D2;
```

Quizz 7: Is there a compilation error?

```
type Date_Acc is access Date;

D1 : Date_Acc := new Date;
D2 : Date_Acc;

D1 := D2;
```

Quizz 8: Is there a compilation error?

```
type String_Acc is access String;

S : String_Acc := new String'("Hello");
C : Character;

C := S.all (0);
```

Quizz 9: Is there a compilation error?

```
type String_Acc is access String;
S : String_Acc := new String'("Hello");
C : Character;

C := S.all (1);
```

Quizz 10: Is there a compilation error?

```
type String_Acc is access String;
S : String_Acc := new String'("Hello");
C : Character;

C := S (1);
```

Quizz 11: Is there a compilation error?

```
type Node;
type Node_Acc is access Node;
type Op_Kind is (Bin_Op, Un_Op);

type Node (Op : Op_Kind) is record
   Id : Natural;
   case Op is
      when Un_Op =>
         Operand : Node_Acc;
      when Bin_Op =>
         Left, Right : Node_Acc;
   end case;
end record;

N : Node (Un_Op);
```

Quizz 12: Is there a compilation error?

```
type Node;
type Node_Acc is access Node;
type Op_Kind is (Bin_Op, Un_Op);

type Node (Op : Op_Kind) is record
   Id : Natural;
   case Op is
      when Un_Op =>
         Operand : Node_Acc;
      when Bin_Op =>
         Left, Right : Node_Acc;
   end case;
end record;

N : Node := (Un_Op, 2, null);
```

Quizz 13: Is there a compilation error?

```
type Node;
type Node_Acc is access Node;
type Op_Kind is (Bin_Op, Un_Op);

type Node (Op : Op_Kind) is record
   Id : Natural;
   case Op is
      when Un_Op =>
         Operand : Node_Acc;
      when Bin_Op =>
         Left, Right : Node_Acc;
   end case;
end record;

N : Node (Un_Op);

N.Left.Id := 12;
```

Quizz 14: Is there a compilation error?

```
type Node;
type Node_Acc is access Node;
type Op_Kind is (Bin_Op, Un_Op);

type Node (Op : Op_Kind) is record
   Id : Natural;
   case Op is
      when Un_Op =>
         Operand : Node_Acc;
      when Bin_Op =>
         Left, Right : Node_Acc;
   end case;
end record;

N : Node_Acc := ...

Put_Line (N.Left.Op'Image);
```

Privacy

Private part

```
package Stacks is
   procedure Hello;

private

   procedure Hello2;
   --  Not visible from external units
end Stacks;
```

Abstract data types: Declaration

```
package Stacks is
   type Stack is private;
   --  Declare a private type: You cannot depend on its
   --  implementation. You can only assign and test for
   --  equality.

   procedure Push (S : in out Stack; Val : Integer);
   procedure Pop (S : in out Stack; Val : out Integer);
private

   subtype Stack_Index is Natural range 1 .. 10;
   type Content_Type is array (Stack_Index) of Natural;

   type Stack is record
      Top : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;
```

Abstract data types: vocabulary

```
package Stacks is
   type Stack is private;
   --  Partial view

   procedure Push (S : in out Stack; Val : Integer);
   procedure Pop (S : in out Stack; Val : out Integer);
private

   subtype Stack_Index is Natural range 1 .. 10;
   type Content_Type is array (Stack_Index) of Natural;

   type Stack is record
      Top     : Stack_Index;
      Content : Content_Type;
   --  Full view
   end record;
end Stacks;
```

Abstract data types

```
--  No need to read the private part to use the package
package Stacks is
   type Stack is private;

   procedure Push (S : in out Stack; Val : Integer);
   procedure Pop (S : in out Stack; Val : out Integer);
private
   ...
end Stacks;
```

```
--  Example of use
with Stacks; use Stacks;

procedure Test_Stack is
   S : Stack;
   Res : Integer;
begin
   Push (S, 5);
   Push (S, 7);
   Pop (S, Res);
end Test_Stack;
```

Limited types

```
package Stacks is
   type Stack is limited private;
   --  Limited type. Cannot assign nor compare.

   procedure Push (S : in out Stack; Val : Integer);
   procedure Pop (S : in out Stack; Val : out Integer);
private
   subtype Stack_Index is Natural range 1 .. 10;
   type Content_Type is array (Stack_Index) of Natural;

   type Stack is limited record
      Top     : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;
```

Limited types

```
package Stacks is
   type Stack is limited private;
   ...
private
   type Stack is record  -- Full view is not limited
      ...
   end record;
end Stacks;
```

```
package Stacks is
   type Stack is limited private;
   ...
private
   ...
   type Stack is limited record -- Full view is limited
      ...
   end record;
end Stacks;
```

Quizz: Privacy

Quizz 1: Is there a compilation error?

```
package Stacks is
   type Stack;
   procedure Push (S : in out Stack; Val : Integer);
   private
   subtype Stack_Index is Natural range 1 .. 10;
   type Content_Type is array (Stack_Index) of Natural;
   type Stack is record
      Top : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;
```

Quizz 2: Is there a compilation error?

```
package Stacks is
   type Stack is private;
   procedure Push (S : in out Stack; Val : Integer);
   private
   type Stack is range 1 .. 100;
end Stacks;
```

Quizz 3: Is there a compilation error?

```
package Stacks is
   type Stack is private;
   procedure Push (S : in out Stack; Val : Integer);
end Stacks;
```

Quizz 4: Is there a compilation error?

```
--  stacks.ads
package Stacks is
   type Stack is private;
   procedure Push (S : in out Stack; Val : Integer);
   private
   type Stack is range 1 .. 100;
end Stacks;

--  test.adb
with Stacks; use Stacks;
procedure Test is
   T : Stack;
begin
   T := 3;
end Test;
```

Quizz 5: Is there a compilation error?

```
--  stacks.ads
package Stacks is
   type Stack is private;
   procedure Push (S : in out Stack; Val : Integer);
   private
   type Stack is range 1 .. 100;
end Stacks;

--  stacks2.ads
with Stacks; use Stacks;

package Stacks2 is
   type Stack2 is record
      S1 : Stack;
      S2 : Stack;
   end record;
end Stacks2;
```

Quizz 6: Is there a compilation error?

```
--  stacks.ads
package Stacks is
   type Stack is limited private;
   procedure Push (S : in out Stack; Val : Integer);
   private
   type Stack is range 1 .. 100;
end Stacks;

--  test.adb
with Stacks; use Stacks;
procedure Test is
   T : Stack := 3;
begin
   ...
end Test;
```

Quizz 7: Is there a compilation error?

```
--  stacks.ads
package Stacks is
   type Stack is limited private;
   procedure Push (S : in out Stack; Val : Integer);
   function Init return Stack;
private
   ...
end Stacks;

--  test.adb
with Stacks; use Stacks;
procedure Test is
   T : Stack := Init;
begin
   ...
end Test;
```

Quizz 8: Is there a compilation error?

```
--  stacks.ads
package Stacks is
   type Stack is limited private;
   procedure Push (S : in out Stack; Val : Integer);
   function Init return Stack;
private
   ...
end Stacks;

--  test.adb
with Stacks; use Stacks;
procedure Test is
   T : Stack;
begin
   T := Init;
   ...
end Test;
```

Quizz 9: Is there a compilation error?

```
--  stacks.ads
package Stacks is
   type Stack is limited private;
   procedure Push (S : in out Stack; Val : Integer);
   procedure Init (S : out Stack);
private
   ...
end Stacks;

--  test.adb
with Stacks; use Stacks;
procedure Test is
   T : Stack;
begin
   Init (T);
   ...
end Test;
```

Quizz 10: Is there a compilation error?

```
--  stacks.ads
package Stacks is
   type Stack is limited private;
   procedure Push (S : in out Stack; Val : Integer);
   procedure Init (S : out Stack);
private
   subtype Stack_Index is Natural range 1 .. 10;
   type Content_Type is array (Stack_Index) of Natural;
   type Stack is record
      Top : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;

--  stacks.adb
package body Stacks is
   procedure Init (S : out Stack) is
   begin
      S := (Top => 1, Content => (others => <>));
   end Init;
   ...
end Stacks;
```

Quizz 11: Is there a compilation error?

```
--  stacks.ads
package Stacks is
   type Stack is limited private;
   procedure Push (S : in out Stack; Val : Integer);
   procedure Init (S : out Stack);
private
   subtype Stack_Index is Natural range 1 .. 10;
   type Content_Type is array (Stack_Index) of Natural;
   type Stack is limited record
      Top : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;

--  stacks.adb
package body Stacks is
   procedure Init (S : out Stack) is
   begin
      S := (Top => 1, Content => (others => <>));
   end Init;
   ...
end Stacks;
```

Quizz 12: Is there a compilation error?

```
package P1 is
   type Stack is limited private;
   ...
end P1;

with P1;
package P2 is
   type T2 is record
      A : P1.Stack;
   end record;
end P2;

with P2; use P2;
...
   V1, V2 : T2;
...
   V2 := V1;
...
```

Generics

Generic declaration

```
--  A generic subprogram is not a subprogram
generic
   --  Formal part
   type Elem is private;
procedure Exchange (A, B: in out Elem);

generic
   type Item is private;
   with function "*" (A, B : Item) return Item is <>;
function Squaring (X : Item) return Item;

--  A generic package is not a package
generic
   type Item is private;
package My_Pkg is
   procedure Exchange (A, B: in out Elem);
end My_Pkg;

--  Only packages and subprograms can be generic. Not types !
```

Generic body

```
procedure Exchange (A, B: in out Elem) is
   T : Elem := A;
begin
   A := B;
   B := T;
end Exchange;
```

Generic instantiation

```
--  declare block: Introduces a declarative part in a
--  statements part
declare
   procedure Int_Exchange is new Exchange (Integer);

   A, B : Integer;
begin
   Int_Exchange (A, B);
end;
```

Formal type

- Validity of the body is checked against the spec, not against the uses (not like C++)
- Not all operators are available with all types
- A formal type specifies the kind of types

```
type T (<>) is limited private; --  Any type
type T is limited private; --  Any definite type
type T (<>) is private; --  Any non-limited type
type T is private; --  Any non-limited definite type (most used)
type T is (<>); --  Discrete types (enum, int, modular)
type T is range <>; --  Signed integer types
type T is mod <>; --  Modular types
type T is digits <>; --  Floating point types
type T is delta <>; --  Fixed point types
type T is array ...  --  Array type
type T is access ...  --  Access type
```

Formal type

Examples

```
type Item is private;
type Index is (<>);
type Vector is array (Index range <>) of Item;
type Link is access Item;
```

Formal objects

```
generic
   type Element_Type is private;
   Max_Size : Integer;
   --  This is a formal object
package Stacks is
   ...
end Stacks;
```

Formal subprograms

```
generic
   type Element_Type is private;
   with function Less_Than (L, R: Element_Type) return Boolean;
   --  This is a formal subprogram. Expands the operation
   --  you can do on Element_Type.
package Ordered_Maps is
    type Ordered_Map is private;
    ...
end Stacks;
```

Quizz

Quizz 1: Is there a compilation error?

```
generic
   type Elem is private;
procedure P;

procedure P1 is new P (Elem => String);
```

Quizz 2: Is there a compilation error?

```
generic
   type Elem (<>) is private;
procedure P;

procedure P is
   Var : Elem;
begin
   null;
end P;
```

Quizz 3: Is there a compilation error?

```
--  p.ads
generic
   type Elem is private;
procedure P;

--  p.adb
procedure P is
   Var : Elem;
begin
   null;
end P;

--  main.adb
with P;

procedure Main is
    procedure Str_P is new P (String);
begin
   null;
end P;
```

Quizz 4: Is there a compilation error?

```
--  p.ads
generic
   type Elem is private;
procedure P;

--  p.adb
procedure P is
   Var : Elem;
begin
   null;
end P;

--  main.adb
with P;

procedure Main is
    procedure Str_P is new P (String (1 .. 10));
begin
   null;
end P;
```

Quizz 5: Is there a compilation error ?

```
--  g.ads
generic
   type T is private;
package G is
   V : T;
end G;

--  p.adb
with G;

procedure P is
   type My_Integer is new Integer;

   package I1 is new G (Integer);
   package I2 is new G (My_Integer);

   use I1, I2;
begin
   V := 0;
end P;
```

Quizz 6: Is there a compilation error ?

```
--  g.ads
generic
   type T is private;
package G is
   V : T;
end G;

--  p.adb
with G;

procedure P is
   type My_Integer is new Integer;

   package I1 is new G (Integer);
   package I2 is new G (My_Integer);

   use I1;
begin
   V := 0;
end P;
```

Quizz 7: Is there a compilation error ?

```
generic
   type Element_Type is private;
procedure P (El : Element_Type);


procedure P (El : Element_Type) is
begin
   Put_Line ("El = " & Element_Type'Image (El));
end P;
```

Exceptions

Exception declaration

```
My_Except : exception;
```

- Like an object. *NOT* a type !

Raising an exception

```
raise My_Except;
--  Execution of current control flow abandoned
```

Handling an exception

```
--  Block (sequence of statements)
begin
   Open (File, In_File, “input.txt”);
exception
   when E : Name_Error =>
   --       ^ Exception to be handled
      Put ("Cannot open input file : ");
      Put_Line (Exception_Message (E));
      raise;
      --  Reraise current occurence
end;
```

Handling an exception

```
procedure Main is
begin
   Open (File, In_File, “input.txt”);
--  Exception block can be added to any block
exception
   when Name_Error =>
      Put (“Cannot open input file“);
end;
```

Handling an exception

```
procedure Main is
begin
   Open (File, In_File, “input.txt”);
exception
   when Name_Error | Constraint_Error =>
   --   ^ Can handle several occurences in the same block
   when others =>
   --   ^ Catch-all
end;
```

Predefined exceptions

- `Constraint_Error`
  - raised when bounds or subtype doesn't match
  - raised in case of overflow (-gnato for GNAT)
  - null dereferenced
  - division by 0
- `Program_Error`
  - weird stuff (eg: elaboration, erroneous execution)
- `Storage_Error`
  - not enough memory (allocator)
  - not enough stack
- `Tasking_Error`

Quizz: Exceptions

Quizz 1: Is there a compilation error

```
procedure P is
   Ex : exception;
begin
   raise Ex;
end;
```

Quizz 2: What will be printed

```
with Text_IO; use Text_IO;
procedure E is
begin
   declare
      A : Positive;
   begin
      A := -5;
   exception
      when Constraint_Error =>
         Put_Line ("caught it");
   end;
exception
   when others =>
      Put_Line ("last chance handler");
end;
```

Quizz 3: What will be printed

```
with Text_IO; use Text_IO;
procedure E is
begin
   declare
      A : Positive;
   begin
      A := -5;
   exception
      when Constraint_Error =>
         Put_Line ("caught it");
         raise;
   end;
exception
   when others =>
      Put_Line ("last chance handler");
end;
```

Quizz 4: What will be printed

```
with Text_IO; use Text_IO;
procedure E is
begin
   declare
      A : Positive := -1;
   begin
      A := -5;
   exception
      when Constraint_Error =>
         Put_Line ("caught it");
   end;
exception
   when others =>
      Put_Line ("last chance handler");
end;
```

Quizz 4: What will be printed

```
with Text_IO; use Text_IO;
procedure E is
begin
   declare
      A, B, C : Positive;
   begin
      A := 10;
      B := 9;
      C := 2;
      A := B - A + C;
   exception
      when Constraint_Error =>
         Put_Line ("caught it");
   end;
exception
   when others =>
      Put_Line ("last chance handler");
end;
```

Tasking

Simple task

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   task T;

   task body T is
   begin
      Put_Line ("In task T");
   end;
begin
   Put_Line ("In main");
end Main;
```

Simple synchronization

```
procedure P is
   task T;
   task body T is
   begin
      for I in 1 .. 10 loop
         Put_Line (“hello”);
      end loop;
   end;
begin
   null;
   --  Will wait here until all tasks have terminated
end;
```

Simple synchronization

```
procedure P is
   task T;
   task body T is
   begin
      for I in 1 .. 10 loop
         Put_Line (“hello”);
      end loop;
   end;
begin
   null;
   --  Will wait here until all tasks have terminated
end;
```

Simple synchronization

```
--  p.ads
package P is
   task T;
end P;

--  p.adb
package body P is
   task body T is
   begin
      for I in 1 .. 10 loop
         Put_Line (“hello”);
      end loop;
   end;
end;

--  main.adb
with P;
procedure Main is
begin
   null;
   --  Will wait here until all tasks have terminated
end;
```

Delay

```
task T;

task body T is
begin
   for I in 1 .. 10 loop
      Put_Line (“hello”);
      delay 1.0;
      --    ^ Wait 1.0 seconds
   end loop;
end;
```

Synchronization: rendez-vous

```
task T is
   entry Start;
end T;

task body T is
begin
   accept Start; -- Waiting for somebody to call the entry
   Put_Line ("In T");
end T;

procedure Main is
begin
   Put_Line ("In Main");
   T.Start --  Calling T's entry
end Main;
```

Synchronization: rendez-vous

```
task T is
   entry Start;
end T;

task body T is
begin
   accept Start; -- Waiting for somebody to call the entry
   Put_Line ("In T");
end T;

procedure Main is
begin
   Put_Line ("In Main");
   T.Start --  Calling T's entry
end Main;
```

Synchronization: rendez-vous

```
task T is
   entry Start;
end T;

task body T is
begin
   loop
      accept Start;
      Put_Line ("In T's loop");
   end loop;
end T;
```

Synchronization: rendez-vous

```
procedure Main is
   task T is
      entry Start (M : String);
      --           ^ Entry parameter
   end T;

   task T1;

   task body T is
   begin
      accept Start (M : String) do
         Put_Line (M);
      end Start;
   end T;

   task body T1 is
   begin
      T.Start ("Hello");
      --       ^ Pass parameter to entry
   end;
begin
   null;
end Main;
```

Cycling tasks

```
with Ada.Real_Time; use Ada.Real_Time;

procedure Main is
   task T;

   task body T is
      Next : Time := Clock;
      Cycle : constant Time_Span := Milliseconds (100);
   begin
      while True loop
         delay until Next;
         Next := Next + Cycle;
      end loop;
   end;
begin
   null;
end Main;
```

Protected objects

Provides Exclusive access/mutual exclusion

```
protected Obj is
   --  Operations go here (only subprograms)
   procedure Set (V: Integer);
   function Get return Integer;
private
   --  Data goes here
   Local : Integer;
end Obj;
```

Protected objects: body

Provides Exclusive access/mutual exclusion

```
protected Obj is
   procedure Set (V: Integer);
   function Get return Integer;
private
   Local : Integer;
end Obj;

protected body Obj is
   --  procedures can modify the data
   procedure Set (V: Integer) is
   begin
      Local := V;
   end Set;

   --  functions cannot modify the data
   function Get return Integer is
   begin
      return Local;
   end Get;
end Obj;
```

Protected objects: entries

```
protected Obj is
   procedure Set (V: Integer);
   entry Get (V : out Integer);
private
   Value : Integer;
   Is_Set : Boolean := False;
end Obj;

protected body Obj is
   procedure Set (V: Integer) is
   begin
      Local := V;
      Is_Set := True;
   end Set;

   entry Get (V : out Integer) when Is_Set is  --  Barrier
   begin
      V := Local;
      Is_Set := False;
   end Get;
end Obj;
```

Protected objects: entries

```
protected body Obj is
   procedure Set (V: Integer) is
   begin
      Local := V;
      Is_Set := True;
   end Set;

   entry Get (V : out Integer)
      when Is_Set is
      -- Entry will be blocked until the condition is true.
      -- Barrier is evaluated at call of entry, and at exit of
      -- procedures and entries.
      -- Calling task will sleep until the barrier is relieved
   begin
      V := Local;
      Is_Set := False;
   end Get;
end Obj;
```

Protected types

```
protected type Obj is
   procedure Set (V: Integer);
   function Get return Integer;
   entry Get_Non_Zero (V : out Integer);
private
   Local : Integer;
end Obj;
```

Quizz

Quizz 1: Is there a compilation error ?

```
task type T;
...
type T_array is array (Natural range <>) of T;
```

Quizz 2: Is there a compilation error ?

```
task type T;
...
type Rec is record
   N : Natural;
   P : T;
end record;

P1, P2: Rec;
...
P1 := P2;
```

Quizz 3: Does this code terminate ?

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   Ok : Boolean := False;
   protected O is
      entry P;
   end O;

   protected body O
      entry P when Ok is
      begin
         Put_Line ("OK");
      end P;
   end O;

   task T;

   task body T is begin
      delay 1.0;
      Ok := True;
   end T;
begin
   O.P;
end;
```

Quizz 4: Does this code terminate ?

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   Ok : Boolean := False;
   protected O is
      entry P;
      procedure P2;
   end O;
   protected body O is
      entry P when Ok is begin
         Put_Line ("OK");
      end P;
      procedure P2 is begin
         null;
      end P2;
   end O;
   task T;
   task body T is begin
      delay 1.0;
      Ok := True;
      O.P2;
   end T;
begin
   O.P;
end;
```

Quizz 5: How does this code terminate ?

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
   task T is
      entry Start;
   end T;

   task body T is
   begin
      accept Start;
      Put_Line ("I'm out");
   end T;
begin
   T.Start;
   T.Start;
end Main;
```

Quizz 6: When does this procedure terminate ?

```
procedure Main is
   task type T;
   task body T is
   begin
      delay 2.0;
   end T;
   type T_Acc is access T;
   T1 : T_Acc;
begin
   T1 := new T;
end Main;
```

Quizz 7: What does this code print ?

```
procedure Main is
   task T is
      entry Start;

   task body T is
   begin
      accept Start do
         Put_Line ("In start");
      end Start;
      Put_Line ("Out of start");
   end T;
begin
   Put_Line ("In main");
   T.Start;
   Put_Line ("In main 2");
end Main;
```

Quizz 8: Is there a compilation error ?

```
procedure Main is
   Ok : Boolean := False;

   protected O is
      function F return Boolean;
   end O;

   protected body O is
      function F return Boolean is
      begin
         Ok := not Ok;
         return Ok;
      end F;
   end O;

   V : Boolean;
begin
   V := O.F;
end;
```

Quizz 9: Is there a compilation error ?

```
procedure Main is
   protected O is
      function F return Boolean;
   private
      Ok : Boolean := False;
   end O;

   protected body O is
      function F return Boolean is
      begin
         Ok := not Ok;
         return Ok;
      end F;
   end O;

   V : Boolean;
begin
   V := O.F;
end;
```

Interfacing

Type convention

```
type Enum is (A, B, C);
pragma Convention (C, Enum);
--                 ^ Use C convention for Enum
```

Type convention

```
with Interfaces.C; use Interfaces.C;
--  Provides C-compatible declarations

type C_Type is record
   A : int;
   B : long;
   C : unsigned;
end record;
```

Foreign subprograms

```
int my_func (int a);
```

```
with Interfaces.C; use Interfaces.C;

function my_func (a : int) return int
  with Import        => True,
       Convention    => C;
--  Imports function 'my_func' from C. You can now call it from Ada.
```

Foreign subprograms

```
void my___func (int a);
```

```
with Interfaces.C; use Interfaces.C;

procedure my_func (a : int)
  with Import        => True,
       Convention    => C,
       External_Name => "my___func";
--  Imports function 'my___func' from C.
```

Foreign subprograms

```
-- c_api.ads
with Interfaces.C; use Interfaces.C;

package C_API is
   procedure My_Func (a : int)
     with Export => True, Convention => C, External_Name => "my_func";
end C_API;

-- c_api.adb
package body C_API is
   procedure My_Func (a : int) is
   begin
      Put_Line (int'Image (a));
   end My_Func;
end C_API;
```

```
extern void my_func (int a);
```

Foreign variables

```
extern int my_var;
```

```
with Interfaces.C; use Interfaces.C;

my_var : int;
pragma Export (C, my_var);
```

Multi-language project

```
project Multilang is

   for Languages use ("ada", "c");

   for Source_Dirs use ("src");
   for Main use ("main.adb");
   for Object_Dir use "obj";

end Multilang;
```

Quizz

Quizz 1: Is there a compilation error

```
procedure P;
   with Import => True,
        Convention => C;

procedure P is
begin
   null;
end;
```

Quizz 2: Is there a compilation error

```
procedure P
   with Export => True, Convention => C
is
begin
   null;
end;
```

Quizz 3: Is there a compilation error

```
procedure P is
   procedure P1;
     with Export => True, Convention => C;

   procedure P1 is
   begin
      null;
   end P1;

begin
   null;
end;
```

Quizz 4: Is there a compilation error

```
function Get_Version return String
  with Import => True, Convention => C;
```

Quizz 5: Is there a compilation error

```
procedure Put_Str (S : String)
  with Import => True, Convention => C;
```

Low level

Deallocation

```
with Ada.Unchecked_Deallocation;

procedure Dealloc is
   type My_Acc is access My_Type;
   procedure Deallocate is new
   Ada.Unchecked_Deallocation (My_Type, My_Acc);
   V : My_Acc;
begin
   V := new My_Type;
   Deallocate (V);
   --  Release the memory, and set V to null (noop if V is already null)
end Dealloc;
```

Querying address

```
with System; use System;

package P is
   V : Integer;
   V_Addr : Address := V'Address;
   --                  ^ Address of V
end P;
```

Querying alignment

```
package P is
   V : Integer;
   V_Alignment : Integer := V'Alignment;
   --                       ^ Alignment of V
end P;
```

Querying size

```
package P is
   V : Integer;

   Typ_Size : Integer := Integer'Size;
   --                    ^ Minimum size for an Integer, in
   --                      bits (== 32)

   Typ_Size : Integer := Natural'Size;
   --                    ^ Minimum size for a Natural, in
   --                      bits (== 31)

end P;
```

Querying size

```
package P is
   Instance_Size : Integer := V'Size;
   --                         ^ Actual size, in bits

   Typ_Size : Integer := Integer'Max_Size_In_Storage_Elements;
   --                  ^ Max size, in storage elements,
   --                    used to store an Int
   --                    (storage element -> byte)
end P;
```

Specifying address

```
with System; use System;

package P is
   V : Integer;

   for V'Address
   use System.Storage_Elements.To_Address (16#fff_0000#);
   --                                      ^ Must be correctly aligned
end P;
```

Specifying address

```
with System; use System;

package P is
   V : Integer
   with Address => System'To_Address (16#fff_0000#);
   --                    ^ GNAT specific attribute

   pragma Import (Ada, V);
   -- Prevents initialization
end P;
```

Specifying address

```
procedure Pouet is
   A : array (1 .. 32) of Integer;

   B : array (1 .. 32 * 4) of Character
   with Address => A'Address;
   --  B is now an overlay for A, except you manipulate
   --  memory in bytes.

   type Rec is record
      A, B : Integer;
   end Rec;

   Inst : Rec;

   C : Integer
   with Address => Inst'Address;
begin
   null;
end Pouet;
```

Specifying size

```
with System; use System;

package P is
   V : Natural
   with Size => 32;
   --           ^ Must be large enough.
   --           ^ Compiler can choose bigger size.
end P;
```

Specifying alignment

```
with System; use System;

package P is
   V : Integer;
   with Alignment => 1;
   --                ^ Address must be a multiple of this.
   --                  So compiler can over align.
end P;
```

Packing arrays

```
procedure BV is
   type Bit_Vector is array (0 .. 31) of Boolean;
   pragma Pack (Bit_Vector);

   B : Bit_Vector;
begin
   Put_Line (Integer'Image (B'Size));
   -- Prints 32
end;
```

Packing records

```
procedure Packed_Rec is
   type My_Rec is record
      A : Boolean;
      C : Natural;
   end record
   with Pack;

   R : My_Rec;
begin
   Put_Line (Integer'Image (R'Size));
   -- Prints 32
end Packed_Rec;
```

Specifying record layout

```
type Register is range 0 .. 15;
   with Size => 4;
--  Size on type only affects components

type Opcode is (Load, Inc, Dec, ..., Mov);
   with Size => 8;

type RR_370_Instruction is record
   Code : Opcode;
   R1   : Register;
   R2   : Register;
end record;

for RR_370_Instruction use record
   Code at 0 range 0 .. 7;
   R1 at 1 range 0 .. 3;
   R2 at 1 range 4 .. 7;
end record;
```

Bit to bit conversion

```
with Ada.Unchecked_Conversion;

procedure Unconv is
   subtype Str4 is String (1 .. 4);
   function To_Str4 is new Ada.Unchecked_Conversion (Integer, Str4);

   V : Integer;
   S : Str4;
   S := To_Str4 (V)
begin
   null;
end Unconv;
```

Pragma Volatile

```
V : Integer;
pragma Volatile (V);
```

Pragma Atomic

```
V : Integer;
pragma Atomic (V);
```

Pragma Inline

```
package P is
   procedure Proc (A : Integer);
   pragma Inline (Proc);
   --  Compiler can read the body
end P;
```

Object oriented programming

Classes

```
package P is
   type My_Class is tagged null record;
   --  Just like a regular record, but with tagged qualifier

   --  Methods are outside of the type definition

   procedure Do_Something (Self : in out My_Class);
end P;
```

Classes

```
package P is
   type My_Class is tagged null record;

   type Derived is new My_Class with record
       A, B : Integer;
       --  You can add field in derived types.
   end record;
end P;
```

Methods

```
package P is
   type My_Class is tagged record
      Id : Integer;
   end record;

   procedure Foo (Self : My_Class);
   --  If you define a procedure taking a My_Class argument,
   --  in the same package, it will be a method.

   type Derived is new My_Class with null record;

   overriding procedure Foo (Self : My_Class);
   --  overriding qualifier is optional, but if it is here,
   --  it must be valid.
end P;
```

Dispatching calls

```
with P; use P;

procedure Main is
   Instance   : My_Class;
   Instance_2 : Derived;
begin
   Foo (Instance);
   --  Static (non dispatching) call to Foo of My_Class

   Foo (Instance_2);
   --  Static (non dispatching) call to Foo of Derived
end Main;
```

Dispatching calls

```
with P; use P;

procedure Main is
   Instance   : My_Class'Class := My_Class'(12);
   --           ^ Denotes the classwide type
   --             Needs to be initialized
   --
   --  Classwide type can be My_Class or any descendent of
   --  My_Class
   Instance_2 : My_Class'Class := Derived'(12);
begin
   Foo (Instance);
   --  Dynamic (dispatching) call to Foo of My_Class

   Foo (Instance_2);
   --  Dynamic (dispatching) call to Foo of Derived
end Main;
```

Dispatching calls

```
with P; use P;

procedure Main is
   Instance   : My_Class'Class := My_Class'(12);
   Instance_2 : My_Class'Class := Derived'(12);
begin
   Foo (Instance);
   --  Dynamic (dispatching) call to Foo of My_Class

   Foo (Instance_2);
   --  Dynamic (dispatching) call to Foo of Derived
end Main;
```

Conversions

```
with P; use P;

procedure Main is
   Instance   : Derived'Class := Derived'(12);
   Instance_2 : My_Class'Class := Instance;
   -- Implicit conversion from Derived'Class to My_Class'Class
   Instance   : My_Class := My_Class (Instance_2);
   --                       ^ Can convert from 'Class to definite
   Instance_2 : Derived;
begin
   Instance := My_Class (Instance_2);
   --          ^ Explicit conversion from definite derived
   --            object to definite My_Class (called view
   --            conversion)
   Instance_2 := Derived (Instance);
   --            ^ COMPILE ERROR, from definite base to definite subclass
   declare
      D : Derived'Class := Derived'Class (Instance_2);
      --                   ^ From classwide base to classwide subclasss
   begin
      null;
   end;
end Main;
```

Dot notation

```
with P;

procedure Main is
   Instance   : P.My_Class'Class := My_Class'(12);
begin
   Instance.Foo;
   --  Call to procedure Foo, with dot notation.
   --  Procedure is visible even though not in scope.
end Main;
```

Quizz: Object oriented programming

Quizz 1: Is there a compilation error?

```
-- p.ads
package P is
   type T is tagged null record;
   procedure Proc (V : T);
end P;

-- main.adb
with P;
procedure Main is
   V : P.T;
begin
   Proc (V);
   V.Proc;
end Main;
```

Quizz 2: Is there a compilation error?

```
package P is
   type T1 is record
      F1 : Integer;
   end record;

   type T2 is new T1 with record
      F2 : Integer;
   end record;
end P;
```

Quizz 3: Is there a compilation error?

```
package P is
   type T1 is range 1 .. 10;
   procedure Proc (V : T1);

   type T2 is new T1;

   type T3 is new T2;
   overriding procedure Proc (V : T3);
end P;
```

Quizz 4: Who is called ?

```
--  pck.ads
package Pck is
   type Root is tagged null record;
   procedure P (V : Root);

   type Child is new Root with null record;
   overriding procedure P (V : Child);

   type Grand_Child is new Child with null record;
   overriding procedure P (V : Grand_Child);
end Pck;

--  main.adb
with Pck;
procedure Main is
   V : Pck.Child;
begin
   V.P;
end;
```

Quizz 5: Who is called ?

```
--  pck.ads
package Pck is
   type Root is tagged null record;
   procedure P (V : Root);

   type Child is new Root with null record;
   overriding procedure P (V : Child);

   type Grand_Child is new Child with null record;
   overriding procedure P (V : Grand_Child);
end Pck;

--  main.adb
with Pck;
procedure Main is
   V : Child'Class := Grand_Child'(others => <>);
begin
   V.P;
end;
```

Quizz 6: Who is called ?

```
--  pck.ads
package Pck is
   type Root is tagged null record;
   procedure P (V : Root);

   type Child is new Root with null record;
   overriding procedure P (V : Child);

   type Grand_Child is new Child with null record;
   overriding procedure P (V : Grand_Child);
end Pck;

--  main.adb
with Pck;
procedure Main is
   W : Grand_Child;
   V : Child := Child (W);
begin
   V.P;
end;
```

Quizz 7: Who is called ?

```
--  pck2.ads
with Pck; use Pck;
package body Pck2 is
   procedure Call (V : Root) is
   begin
       V.P;
   end Call;
end Pck2;

--  main.adb
with Pck, Pck2; use Pck, Pck2;
procedure Main is
   V : Child;
begin
   Call (Root (V));
end;
```

Quizz 8: Who is called ?

```
--  pck2.adb/s
with Pck; use Pck;
package body Pck2 is
   procedure Call (V : Root'Class) is
   begin
       V.P;
   end Call;
end Pck2;

--  main.adb
with Pck, Pck2; use Pck, Pck2;
procedure Main is
   V : Child;
begin
   Call (V);
end;
```