

Introduction to Ada - Part 2

Raphael Amiard - Adacore
amiard@adacore.com

Generics

```
-- 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 !
```

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

```
-- 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;
```

- 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
```

Examples

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

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



```
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;
```

Quiz 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;
```

Quiz 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

```
My_Except : exception;
```

- Like an object. *NOT* a type !

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

```
-- 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 occurrence
end;
```

```
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;
```

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

- *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;
```

Object oriented programming

```
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;
```



```
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;
```

```
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;
```

```
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;
```

```
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;
```

```
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;
```

```
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 subclasses
  begin
    null;
  end;
end Main;
```

```
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;
```


Low level

```
with System; use System;

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

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

```
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;
```

```
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;
```

```
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;
```

```
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;
```

```
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;
```



```
with System; use System;

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

```
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;
```

```
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;
```

```
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;
```

```
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;
```

```
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;
```

```
type Video_Buffer is array (Natural range <>) of RGB_Value;  
pragma Volatile (Video_Buffer);
```

```
Device_Status : Status_Register;  
pragma Atomic (Device_Status);  
for Device_Status'Address use System.Storage_Elements.To_Address (16#8010_FF74#);
```



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

Tasking

```
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;
```

```
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;
```

```
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;
```

```
-- 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;
```

```
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;
```

```
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;
```



```
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;
```

```
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;
```

```
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;
```

```
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;
```

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 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 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 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;
```

Quiz 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;
```

Quiz 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;
```

Quiz 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;
```