Packages, Generics - intro

Modular programming

- it allows arbitrary declarations in a declarative part
- separation of programs into multiple packages and sub-package
- declare our types and variables in the bodies of main procedures or package specification
- packages let you make your code modular
- with clause indicates a dependency
- accessing entities from a package uses the dot notation
- to make every entity of a package visible directly in the current scope, using use clause

Packages

- **Using a package**
- with Ada.Text_IO; use Ada.Text_IO;
- Put_Line is a subprogram that comes from the Ada.Text_IO
- **Package body**
- Increment_By function has to be declared in the body.
- Last_Increment variable in the body, and make them inaccessible to the user of the Operations package, providing a first form of encapsulation.
- entities declared in the body are *only* visible in the body.

Example

```
package Operations is   --  Declaration
function Increment_By (I: Integer; Incr: Integer := 0) return Integer;
function Get_Increment_Value return Integer;
end Operations;

package body Operations is
Last_Increment : Integer := 1;

function Increment_By (I: Integer; Incr: Integer := 0) return Integer is
begin      if Incr /= 0 then Last_Increment := Incr;      end if;
return I + Last_Increment;   end Increment_By;

function Get_Increment_Value return Integer is
begin      return Last_Increment;   end Get_Increment_Value;
end Operations;
```

Example

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

procedure Main is
I : Integer := 0;
R : Integer;
begin
R := Increment_By (I);
R := Increment_By (I, 10);
end Main;
```

Privacy - Abstract data types

- Encapsulation is the concept that distinguishes between the code's public interface and its private implementation

```
package Stacks is
   type Stack is private; -- You cannot depend on implementation
                          -- You can only assign and test 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;
```

Privacy

- we define a stack type in the public part – visible part
- in the private part, we define the representation of that type
- we can also declare other types that will be used as *helpers* for our main public type
- Stack type as viewed from the public part is called the partial view
- Stack type as viewed from the private part or the body of the package is called the full view of the type
- from the point of view of the client (the *with*'ing unit), only the public (visible) part is important

Usage

```
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

- Ada's *limited type* facility allows you to declare a type for which assignment and comparison operations are not automatically provided.

```
package Stacks is
type Stack is limited private;   --  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

- for some data types the built-in assignment operation might be incorrect
- Ada does allow you to overload the comparison operators = and /= for limited types (and to override the built-in declarations for non-limited types).
- Ada also allows you to implement special semantics for assignment via controlled types
- packages can have child packages

Child packages

- the private part of a package P is meant to encapsulate information,
- certain parts of a child package P.C can have access to this private part of P.
- In those cases, information from the private part of P can then be used as if it were declared in the public part of its specification.
- To be more specific, the body of P.C and the private part of the specification of P.C have access to the private part of P.
- However, the public part of the specification of P.C only has access to the public part of P's specification.

Child packages

```
package Encapsulate is
 procedure Hello;
private
   procedure Hello2;   --  Not visible from external units
                       --  But visible in child packages
end Encapsulate;

with Ada.Text_IO; use Ada.Text_IO;
package body Encapsulate is
  procedure Hello is   begin     Put_Line ("Hello");   end Hello;
  procedure Hello2 is   begin     Put_Line ("Hello #2");   end Hello2;
end Encapsulate;

package Encapsulate.Child is
procedure Hello3;
end Encapsulate.Child;
```

Child packages

```
package Encapsulate is
 procedure Hello;
private
   procedure Hello2;   --  Not visible from external units
  --  But visible in child packages
end Encapsulate;

with Ada.Text_IO; use Ada.Text_IO;
package body Encapsulate is
  procedure Hello is   begin      Put_Line ("Hello");   end Hello;
  procedure Hello2 is   begin      Put_Line ("Hello #2");   end Hello2;
end Encapsulate;

package Encapsulate.Child is
procedure Hello3;
end Encapsulate.Child;
```

Child packages

```
with Ada.Text_IO; use Ada.Text_IO;
package body Encapsulate.Child is
procedure Hello3 is
begin --  Using private proc Hello2 from the  parent package
 Hello2;   Put_Line ("Hello #3");
end Hello3;
end Encapsulate.Child;

with Encapsulate.Child;
procedure Main is
begin
Encapsulate.Child.Hello3;
end Main;
```

Types in child packages

- same mechanism applies to types declared in the private part of a parent package. For instance, the body of a child package can access components of a record declared in the private part of its parent package.

```
package My_Types is
 type Priv_Rec is private;
private
type Priv_Rec is record
 Number : Integer := 42;
end record;
end My_Types;

package My_Types.Ops is
 procedure Display (E : Priv_Rec);
end My_Types.Ops;
```

Types in child packages

```
with Ada.Text_IO; use Ada.Text_IO;
package body My_Types.Ops is
procedure Display (E : Priv_Rec) is
begin     Put_Line ("Priv_Rec.Number: " & Integer'Image (E.Number));
end Display; end My_Types.Ops;

with Ada.Text_IO;  use Ada.Text_IO;with My_Types;
use My_Types;with My_Types.Ops; use My_Types.Ops;
procedure Main is
E : Priv_Rec;
begin   Put_Line ("Presenting information:");
-  The following triggers compilation error, no access to Number component
--  Put_Line ("Priv_Rec.Number: " & Integer'Image (E.Number));
Display (E); -- is defined in My_Types.Ops which is child of My_Types,
                    -- Ops has access to parent private Priv_Rec
end Main;
```

- allows for extending the functionality of a parent package and,
- at the same time, retain its encapsulation

Generic packages

- Generic keyword and formal specifications

```
generic
   type T is private;
package Element is
procedure Set (E : T);
procedure Reset;
function Is_Valid return Boolean;
private
Value : T;
Valid : Boolean := False;
end Element;
```

Example

```
package body Element is
procedure Set (E : T) is
begin
Value := E;
Valid := True;
end Set;
procedure Reset is
begin
Valid := False;
end Reset;
function Is_Valid return Boolean is (Valid);
end Element;
```

Example

```
with Ada.Text_IO; use Ada.Text_IO; with Element;
procedure Show_Generic_Package is
package I is new Element (T => Integer);
procedure Display_Initialized is
begin
 if      I.Is_Valid then        Put_Line ("Value is initialized");
else        Put_Line ("Value is not initialized");      end if;
end Display_Initialized;
begin
Display_Initialized;   Put_Line ("Initializing...");
I.Set (5);   Display_Initialized;

Put_Line ("Reseting...");   I.Reset;
Display_Initialized;
end Show_Generic_Package;
```