Survol du langage Ada

Thierry Lacoste

27 octobre 2007

Ada encourage l'écriture d'**unités** autonomes, compilables séparément et réutilisables.

L'ensemble des unités constitue la **bibliothèque**. La construction d'une application se fait par assemblage d'éléments issus de cette bibliothèque.

Un programme Ada complet est un programme principal (lui-même unité de bibliothèque) appelant les services d'autres unités de bibliothèque.

⋆

Les unités de bibliothèque peuvent être
- des **procédures**,
- des **fonctions**, ou
- des **paquetages**.

Les fonctions et les procédures sont appelées **sous-programmes**.

Un paquetage est un ensemble de ressources de calcul (types, variables, sous-programmes ou encore paquetages) logiquement reliées.

Chapitre 1

Exemple de programme

On veut écrire un programme qui imprime la *racine carrée entière* d'un nombre entier (par exemple 10) ; la racine carrée entière d'un entier n est le plus grand entier r tel que $r^2 \leq n$ (donc la racine entière de 10 est 3).

On suppose que notre bibliothèque contient les unités suivantes :

- `Sqrt` : une fonction d'un paramètre entier qui renvoie la partie entière de sa racine carrée.
- `Io` : un paquetage contenant divers utilitaires d'entrées-sorties permettant de
  - lire des nombres,
  - imprimer des nombres et
  - imprimer des chaînes de caractères.

1.1 Première version

Notre programme principal est une procédure nommée `Root` située dans le fichier root.adb[1] .

1.1.1 La clause `with`

La clause `with` donne les unités de bibliothèque dont notre procédure a besoin.

1.1.2 La clause `use`

En écrivant `use Io` dans `Root`[2] , nous nous donnons un accès direct aux utilitaires du paquetage `Io`. Sans cette clause, nous devons utiliser la *notation pointée* pour indiquer où trouver la procédure `Put`
Attention ! La clause `use` est déconseillée ; le renommage permet de donner des noms courts aux unités utilisées mais évite les problèmes liés à la clause `use` (voire `Root2`[3]).

1.1.3 Appels de sous-programmes et paramètres effectifs

Le programme contient une seule **instruction** : un **appel** à la procédure `Put` avec l'**expression** `Sqrt (10)` comme **paramètre effectif**.

Cette expression est un **appel** à la fonction `Sqrt` avec l'**expression** `10` comme **paramètre effectif**.

⋆

Une procédure ne retourne pas de résultat ; elle est donc appelée en temps qu'instruction : on dit d'un appel de **procédure** qu'il est **exécuté**.

Une fonction retourne un résultat ; elle est donc appelée à l'intérieur d'une expression : on dit d'un appel de **fonction** qu'il est **évalué**.

1.2 Deuxième version

Notre deuxième version de `Root`[4] lit le nombre dont on veut connaître la racine carrée.

1.2.1 Déclarations et instructions

1. Entre `is` et `begin`, se trouvent les **déclarations**.
2. Entre `begin` et `end`, se trouvent les **instructions**.

Les déclarations servent à introduire les entités que les instructions manipulent. Lors de l'exécution d'un programme,

1. les **déclarations** sont **élaborées**, et
2. les **instructions** sont **exécutées**.

Remarque. Tout programme Ada est strictement découpé en

1. zones déclaratives où les instructions sont interdites ;
2. zones instructions où les déclarations sont interdites.

1.2.2 Variables et types

La variable `I` est de type `Integer`, un type prédéfini.

Les valeurs de ce type forment un ensemble d'entiers relatifs : notre variable `I` ne peut prendre sa (ses) valeur(s) que dans cet ensemble.

1.2.3 Identificateurs

Les identificateurs se répartissent en deux catégories :

– Les **mots réservés** comme `procedure` et `is`.

– Les **autres** identificateurs, comme `Put` ou `I`. Certains, comme `Integer`, sont **prédéfinis**, mais nous pouvons changer leur signification.

1.3 Troisième version

Notre troisième version de `Roots`[5] traite une série de nombres.

1.3.1 Paramètres par défaut

L'appel `New_Line (2)` affiche deux sauts de ligne sur la sortie.

La procédure `New_Line` a été écrite avec un **paramètre par défaut** égal à 1. Donc l'appel à `New_Line` sans paramètre est équivalent à l'appel `New_Line (1)` et affiche un saut de ligne sur la sortie.

1.3.2 Surcharge

Les appels à la procédure `Put` se répartissent en deux catégories selon que l'argument est

– un entier (`I` et `Sqrt (I)`) ou

– une chaîne de caractères (`"Racines de nombres divers"` et `" est "`).

Ce sont des appels à des procédures *différentes* ; le compilateur est en mesure de déterminer quelle est la procédure appelée d'après le type de l'argument.

On appelle **surcharge** le fait d'avoir plusieurs procédures de même nom.

1.3.3 Structures de contrôle

Les structures de contrôle sont les **boucles** et les instructions **conditionnelles**.

1. **la boucle :** les instructions qui apparaissent entre `loop` et `end loop` sont répétées jusqu'à ce que la condition `I = 0` dans l'instruction `exit` soit vraie ; lorsque cela arrive, la boucle se termine et l'exécution se poursuit juste après le `end loop`.
2. **la conditionnelle :** l'appel de `Sqrt` n'est effectué que si `X` n'est pas négatif.

Chapitre 2

Unités

Les unités de bibliothèque peuvent être
– des **procédures**,
– des **fonctions**, ou
– des **paquetages**.
Les fonctions et les procédures sont appelés **sous-programmes**.

2.1 Sous-programmes

Rappel : Une procédure est appelée en temps qu'instruction : on dit d'un appel de **procédure** qu'il est **exécuté** car il ne retourne pas de résultat.

Une fonction est appelée à l'intérieur d'une expression : on dit d'un appel de **fonction** qu'il est **évalué** car il retourne un résultat.

2.1.1 Procédures

Notre programme principal est une procédure. Pour obtenir un exécutable, il faut compiler une procédure sans paramètre.

Les procédures peuvent avoir des paramètres ; nous y reviendrons lorsque nous étudirons le paquetage `Io`.

2.1.2 Fonctions

Les fonctions peuvent avoir une **spécification** qui décrit comment l'appeler et et le résultat qu'elle fournit, mais pas comment celui-ci est calculé.

Les fonctions ont obligatoirement un **corps** qui décrit comment le résultat est calculé.

Spécification

La **spécification** de la fonction `Sqrt` se trouve dans le fichier sqrt.ads[6] .
Elle indique le **type** des **paramètres** et du **résultat** de la fonction.

Corps

Le **corps** de la fonction se trouve dans le fichier sqrt.adb[7] .
L'instruction `return` indique le **résultat** de la fonction.

2.1.3 Paramètres effectifs et paramètres formels

Le terme "paramètre" possède deux significations :
- un paramètre dans une spécification de sous-programme est appelé **paramètre formel du sous-programme**, et
- l'argument d'un appel de sous-programme est appelé **paramètre effectif de l'appel au sous-programme**.

Exemples :
- `X` est le paramètre formel de la fonction `Sqrt`,
- `10` est le paramètre effectif de l'appel `Sqrt (10)`, et
- `I` est le paramètre effectif de l'appel `Sqrt (I)`.

2.2 Paquetages

Le paquetage `Io` est en deux parties :

1. la **spécification** qui décrit son interface avec le monde extérieur, et
2. le **corps** qui contient les détails d'implémentation.

2.2.1 Spécification

La spécification du paquetage `Io` se trouve dans le fichier io.ads[8] .
Le paramètre formel de `Get` est un paramètre `out` car c'est un **résultat** da la procédure. L'effet d'un appel à `Get` comme `Get (I)` est de transmettre une valeur de la procédure vers le paramètre effectif `I`.

Les autres paramètres sont tous `in` car ce sont des **données**.

Remarques.

- Seule la **spécification** des procédures figure dans la spécification du paquetage.
- Notons les spécifications **surchargées** de `Put`. Il y a quatre procédures `Put` qui se distinguent par le type de leur paramètre ; ce type est respectivement `Integer`, `Character`, `Float` et `String`.
- Notons enfin comment la valeur par défaut de la procédure `New_Line` est indiquée.

2.2.2 Corps

Le corps du paquetage `Io`, situé dans le fichier io.adb[9] contient les corps complets des procédures ainsi que du matériel supplémentaire **caché** à l'utilisateur du paquetage.

Remarque. La clause `with` montre que l'implémentation des procédures de `Io` utilise le paquetage plus général `Ada.Text_IO`, ainsi que les paquetages `Ada.Integer_Text_IO` et `Ada.Float_Text_IO`.

⋆

Ces paquetages existent réellement ; il sont fournis avec tout compilateur Ada.

Remarque. Un paquetage, nommé `Standard`, contient la déclaration de tous les identificateurs prédéfinis comme `Integer` et `String`. Toute unité Ada possède implicitement un accès direct à ce paquetage.

Chapitre 3

Types et typage fort

3.1 Définition d'un type

1. un ensemble de **valeurs**, et
2. un ensemble d'**opérations** sur ces valeurs.

⋆

Exemple : le type entier `Integer` est défini par
- *les valeurs* : $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$
- *les opérations* : $+, -, *, \ldots$

3.2 Typage fort et contrôle à la compilation

La règle fondamentale du typage fort empêche d'affecter une valeur d'un type à une variable d'un autre type.

Le respect de cette règle est **contrôlé à la compilation** et tout programme ne la respectant pas sera rejetté par le compilateur.

3.2.1 Types énumérés

Le programme enum.adb[10] illustre la règle du typage fort.

La variable `X` ne peut prendre que l'une des trois valeurs `Rouge`, `Orange` ou `Vert`. La variable `B` ne peut prendre que l'une des sept valeurs représentant les jours de la semaine.

Ceci explique pourquoi la dernière affectation est illégale comme le montre le diagnostique enum.log[11] du compilateur.

⋆

Le contrôle sur les types est extrêmement strict et ne se limite pas à l'affectation. En effet, le compilateur interdit toute comparaison entre des objets de types différents. Ainsi, nous ne pourrions pas écrire :

```
if A = X then ...
```

3.2.2 Types prédéfinis

Types énumératifs

Deux types énumératifs sont prédéfinis dans le paquetage `Standard` :

- `type Boolean is (False, True) ;` qui joue un rôle fondamental dans le contrôle du flot des instructions.
  En effet les structures conditionnelles orientent ce flot en choisissant une alternative d'après la valeur d'une expression de type `Boolean`.
  Les opérateurs relationnels (par exemple `<` ou `=`) produisent un résultat de ce type à partir de valeurs d'autres types.

⋆

- `Character` qui joue un rôle important dans les entrées-sorties.
  Les valeurs comprennent les caractères imprimables comme `'x'`, `'A'` ou `'''` ainsi que des caractères de contrôle.

Types numériques

Les autres types de base sont les types numériques.
Les trois classes principales de types numériques sont :

1. les types **entiers** (comme `Integer`) et
2. les types **point-flottant** (comme `Float`).
3. les types **point-fixe**.

Les types point-fixe et point-flottant sont des nombres à virgule habituellement considérés (à tort) comme des réels.

3.3 Intérêt du typage fort

Le typage fort permet de capturer les erreurs de conceptions le plus tôt possible, c'est à dire à la compilation.

3.3.1 Premier exemple

Prenons l'exemple d'un programme qui doit gérer les avoirs d'une banque à la Banque de France (en Francs) et à la Reserve Fédérale américaine (en Dollars).

Ces deux avoirs sont stockés respectivement dans les variables `Account_BF` et `Account_FR`. Dans un langage autre autre que Ada, on serait tenté de les déclarer de type `Integer`, comme dans le programme bank.adb[12] qui comporte une grave erreur de concetion qui peut être difficile à détecter.

3.3.2 Types dérivés

En Ada, le programmeur est encouragé à définir deux types distincts pour les Francs et les Dollars.

On peut par exemple définir ces deux types comme **types dérivés** du type prédéfini `Integer`.

C'est ce que fait le programme bank.adb[13] que le compilateur refuse, comme le montre le diagnostique bank.log[14] .

3.3.3 Conversion

Si le programmeur veut vraiment cette opération, il peut l'obtenir par conversion (*cast* en anglais) comme le montre le programme bank.adb[15] .

3.3.4 Surcharge

L'opération reste illogique ; le programmeur pourra utiliser la surcharge pour définir des fonctions de conversion comme le montre le programme bank.adb[16] .

Chapitre 4

Généricité

Le typage fort est interessant pour la sécurité mais il gène la **réutilisation** du code.

Par exemple, si l'on écrit une procédure de tri pour des tableaux d'entiers, il est impossible de l'utiliser pour trier des tableaux de flottants.

La **généricité** permet de remédier à ce problème en donnant la possibilité au programmeur de *paramètrer* ses unités.

4.1 Exemple

Pour illustrer ce principe, reprenons notre exemple de la banque pour lequel nous ne disposons pas de moyens de saisir et d'afficher des Francs et des Dollars, comme le montre le programme bank.adb[17] et le diagnostique du compilateur bank.log[18] .

Remarque. L'option `-gnatf` du compilateur donne plus de détails, comme le montre bank2.log[19] .

4.2 Solution

Nous utilisons le paquetage `Integer_IO` appartenant au paquetage `Text_IO`. Il s'agit d'un paquetage **générique** pour les types entiers permettant d'obtenir les entrées/sorties sur n'importe quel type entier.

Le programme bank.adb[20] montre l'utilisation de ce paquetage.

4.3 Autre solution

On pourrait intégrer ces entrées/sorties à notre paquetage `Io`, mais ce n'est pas une bonne solution.

Il vaut mieux créer un paquetage dédié à la gestion des différentes monnaies.

Ce paquetage s'appelle `Currencies` ; sa spécification se trouve dans le fichier currencies.ads[21] et son corps dans le fichier currencies.adb[22] .

Le programme bank.adb[23] montre l'utilisation du paquetage `Currencies`.

4.4 Meilleure solution

Une meilleure solution consiste à faire un paquetage `Io` d'entrées/sorties **enfant** du paquetage `Currencies`.

La nouvelle spécification du paquetage `Currencies` se trouve dans le fichier currencies.ads[24] et son corps dans le fichier currencies.adb[25] .

Le paquetage `Currencies.Io` est **enfant** de `Currencies` ; sa spécification se trouve dans le fichier currencies-io.ads[26] et son corps dans le fichier currencies-io.adb[27] .

Le programme bank.adb[28] montre l'utilisation du paquetage `Currencies` ainsi que de son enfant `Io`.

Chapitre 5

Erreurs et exceptions

Notre fonction `Sqrt` renvoie 0 si son argument est négatif. Ce résultat est faux.
Remarque. Aucun résultat ne peut être correct.

Dans le cas où une fonction ne peut pas rendre de valeur, elle **lève une exception** pour signaler que quelque chose d'inhabituel s'est produit ; ce signal brise la suite normale des instructions.

5.1 Levée des exceptions

Dans notre fonction `Sqrt`[29] , nous avons choisi l'exception `Constraint_Error` qui est prédéfinie dans le paquetage `Standard`

L'instruction `raise` permet à la fonction de signaler qu'elle ne peut rien faire avec un argument négatif.

5.2 Effet d'une exception

Si rien n'est prévu dans la procédure `Roots`[30] pour traiter cette exception, notre programme s'arrêtera brutalement.

Il rend la main au système d'exploitation avec un message indiquant la raison de la défaillance.

5.3 Sous-types et contrôle à l'exécution

Nous pouvons laisser au système d'exécution Ada le soin de lever l'exception par le biais des **sous-types**.

Si nous décrivons le paramètre formel de `Sqrt` comme un `Natural` (l'ensemble des `Integer` non négatifs) alors notre programme lèvera l'exception `Constraint_Error` à chaque appel de la fonction `Sqrt` avec un argument négatif (voire `Sqrt`[31]).

Remarques.

- Un sous-type est un **sous-ensemble** des valeurs d'un type et non un nouveau type.
- Le **contrôle du typage** se fait à la **compilation** (les sous-types n'y interviennent pas).
- Le **respect des sous-types** est assurée pendant l'**exécution** du programme.

5.4 Rattrapage des exceptions

Ada nous permet de **rattraper** une exception afin d'effectuer une action corrective lorsqu'elle se produit.

Exemple : Nous pourrions remplacer la conditionnelle

```
if I < 0 then
   Io.Put ("non calculable");
else
   Io.Put (Sqrt (I));
end if;
```

par le **bloc**

```
begin
   Io.Put (Sqrt (I));
exception
   when Constraint_Error => Io.Put ("non calculable");
end;
```

Voyez le programme `Roots`[32] .

Remarque. Dans un cas aussi simple, la conditionnelle est préférable.

$\star$

Un bloc est un cadre composé de
- **déclarations**,
- **instructions**, et
- **traites-exception** qui indiquent les instructions à exécuter dans le cas où une exception s'est produite dans le bloc.

Si une exception est levée pendant l'exécution des instructions située entre `begin` et `exception`, alors

1. les instructions qui suivent le traite-exception correspondant sont exécutées, et le bloc se termine normalement.
2. Si il n'y a pas de traite-exception correspondant à l'exception qui est levée (par exemple, si l'exception `Storage_Error` est levée), alors
   (a) le contrôle remonte la hiérarchie d'imbrication des blocs jusqu'à trouver un traite-exception ad-hoc, ou
   (b) jusqu'au système d'exploitation.

Notes

[1] fichier : Exemple/V1/Ada/root.adb

```
with Sqrt, Io;
procedure Root is
begin
   Io.Put (Sqrt (10));
end Root;
```

[2] fichier : Exemple/V1/Ada/Use/root.adb

```
with Sqrt, Io;
procedure Root is
   use Io;
begin
   Put (Sqrt (10));
end Root;
```

[3] fichier : Exemple/V1/Ada/Use/root2.adb

```
with Sqrt;
with Ada.Integer_Text_IO;
procedure Root2 is
   package IIO renames Ada.Integer_Text_IO;
begin
   IIO.Put (Sqrt (10));
end Root2;
```

[4] fichier : Exemple/V2/Ada/root.adb

```
with Sqrt, Io;
procedure Root is
   I : Integer;
begin
   Io.Get (I);
   Io.Put (Sqrt (I));
end Root;
```

[5] fichier : Exemple/V3/Ada/roots.adb

```
with Sqrt, Io;
procedure Roots is
   I : Integer;
begin
   Io.Put ("Racines  de nombres divers");
   Io.New_Line (2);
   loop
      Io.Put ("Donnez un entier : ");
      Io.Get (I);
      exit when I = 0;
      Io.Put ("La racine de ");
      Io.Put (I);
      Io.Put (" est ");
      if I < 0 then
         Io.Put ("non calculable");
      else
         Io.Put (Sqrt (I));
      end if;
      Io.New_Line;
   end loop;
   Io.New_Line;
   Io.Put ("Fin du programme");
end Roots;
```

[6] fichier : Exemple/V1/Ada/sqrt.ads

```
function Sqrt (X : Integer) return Integer;
```

[7] fichier : Exemple/V1/Ada/sqrt.adb

```
function Sqrt (X : Integer) return Integer is
   R : Integer;
begin
   R := 0;
   while (R + 1) ** 2 <= X loop
      R := R + 1;
   end loop;
   return R;
end Sqrt;
```

[8] fichier : Exemple/V1/Ada/io.ads

```
package Io is

   --  Spécification du paquetage Io.
   --  Elle donne la liste de ce dont dispose l'utilisateur du paquetage.
   --  ================================================================

   procedure Get (I : out Integer); -- saisie d'un entier
   procedure Put (I : in Integer); -- affichage d'un entier

   procedure Get (C : out Character); -- saisie d'un caractère
   procedure Put (C : in Character); -- affichage d'un caractère

   procedure Get (F : out Float); -- saisie d'un nombre flottant
   procedure Put (F : in Float); -- affichage d'un nombre flottant

   procedure Put (S : in String); -- affichage d'une chaine de caractères
   procedure New_Line (N : in Integer := 1); -- passage à la ligne

end Io;
```

[9] fichier : Exemple/V1/Ada/io.adb

```
with Ada.Text_IO;
with Ada.Integer_Text_IO;
with Ada.Float_Text_IO;
package body Io is

   --  Corps du paquetage Io.
   --  Il fournit la réalisation de ce qui est donné dans la spécification.
   --  ================================================================

   --  1) Déclarations locales au corps du paquetage.
   --  Elles sont invisibles de l'utilisateur du paquetage
   --  car elles ne figurent pas dans la spécification.

   package Txt_Io renames Ada.Text_IO;
   package Int_Io renames Ada.Integer_Text_IO;
   package Flt_Io renames Ada.Float_Text_IO;

   --  2) Implémentation (i.e. réalisation) de toutes les procédures
   --  données dans la spécifications.
```

```
   ---------
   -- Get --
   ---------

   procedure Get (I : out Integer) is
   begin
      Int_Io.Get (I);
--      if Txt_Io.End_Of_Line then Txt_Io.Skip_Line; end if;
   end Get;

   ---------
   -- Get --
   ---------

   procedure Get (C : out Character) is
   begin
      Txt_Io.Get (C);
   end Get;

   ---------
   -- Get --
   ---------

   procedure Get (F : out Float) is
   begin
      Flt_Io.Get (F);
   end Get;

   --------------
   -- New_Line --
   --------------

   procedure New_Line (N : in Integer := 1) is
   begin
      for I in Integer range 1 .. N loop
         Txt_Io.New_Line;
      end loop;
   end New_Line;

   ---------
   -- Put --
   ---------

   procedure Put (I : in Integer) is
```

```
   begin
      Int_Io.Put (Item => I, Width => 0);
--      Int_Io.Put (I);
   end Put;

   ---------
   -- Put --
   ---------

   procedure Put (C : in Character) is
   begin
      Txt_Io.Put (C);
   end Put;

   ---------
   -- Put --
   ---------

   procedure Put (F : in Float) is
   begin
      Flt_Io.Put (Item => F, Fore => 0, Aft => 0, Exp => 0);
--      Flt_Io.Put (F);
   end Put;

   ---------
   -- Put --
   ---------

   procedure Put (S : in String) is
   begin
      Txt_Io.Put (S);
   end Put;

end Io;
```

[10] fichier : Types/Ada/enum.adb

```
procedure Enum is

   type Couleur is (Rouge, Orange, Vert);
   type Jour is (Lundi, Mardi, Mercredi, Jeudi,
                 Vendredi, Samedi, Dimanche);
```

```
   X, Y : Couleur;
   A, B : Jour;

begin
   X := Rouge; -- OK
   A := Jeudi; -- OK
   B := X;     -- Illégal
end Enum;
```

[11] fichier : Types/Ada/enum.log

```
$ gnatmake enum
gnatgcc -c enum.adb
enum.adb:13:09: expected type "Jour" defined at line 4
enum.adb:13:09: found type "Couleur" defined at line 3
gnatmake: "enum.adb" compilation error
```

[12] fichier : Types/Ada/Banque/V0/bank.adb

```
with Io;
procedure Bank is

   Account_BF : Integer; -- le compte en Francs
   Account_FR : Integer; -- le compte en Dollars

begin

   Io.Put ("Donnez le montant en Francs : ");
   Io.Get (Account_BF);
   Io.Put ("Donnez le montant en Dollars : ");
   Io.Get (Account_FR);

   --  Transfert de compte

   Account_BF := Account_BF + Account_FR; -- ERREUR !
   Account_FR := 0;

   Io.Put ("Voici le montant en Francs : ");
   Io.Put (Account_BF);
   Io.New_Line;
   Io.Put ("Voici le montant en Dollars : ");
   Io.Put (Account_FR);
   Io.New_Line;
```

```
end Bank;
```

[13] fichier : Types/Ada/Banque/V1/bank.adb

```
procedure Bank is

   type Franc is new Integer;
   type Dollar is new Integer;

   --  Les types Franc et Dollar sont dérivés du type Integer
   --  Ils en pmossèdent toutes les caractèristiques
   --  Mais ils sont distincts

   Account_BF : Franc; -- le compte en Francs
   Account_FR : Dollar; -- le compte en Dollars

begin

   Account_BF := 10;
   Account_FR := 5;

   --  Transfert de compte

   Account_BF := Account_BF + Account_FR; -- ERREUR !
   Account_FR := 0;

end Bank;
```

[14] fichier : Types/Ada/Banque/V1/bank.log

```
$ gnatmake bank
gnatgcc -c bank.adb
bank.adb:20:29: invalid operand types for operator "+"
bank.adb:20:29: left operand has type "Franc" defined at line 3
bank.adb:20:29: right operand has type "Dollar" defined at line 4
gnatmake: "bank.adb" compilation error
```

[15] fichier : Types/Ada/Banque/V2/bank.adb

```
procedure Bank is
```

```
   type Franc is new Integer;
   type Dollar is new Integer;

   --  Les types Franc et Dollar sont dérivés du type Integer
   --  Ils en pmossèdent toutes les caractèristiques
   --  Mais ils sont distincts

   Account_BF : Franc; -- le compte en Francs
   Account_FR : Dollar; -- le compte en Dollars

begin

   Account_BF := 10;
   Account_FR := 5;

   --  Transfert de compte

   Account_BF := Account_BF + Franc (Account_FR); -- Si l'ERREUR est voulue !?!
   Account_FR := 0;

end Bank;
```

[16] fichier : Types/Ada/Banque/V3/bank.adb

```
procedure Bank is

   type Franc is new Integer;
   type Dollar is new Integer;

   Account_BF : Franc; -- le compte en Francs
   Account_FR : Dollar; -- le compte en Dollars

   --  On peut surcharger la fonction Convert qui sert à faire
   --  une vraie conversion ; ce sont les fonctions :

   function Convert (F : Franc) return Dollar;
   function Convert (D : Dollar) return Franc;

   --  Voici leur corps qui utilise la constante Rate
   --  comme taux de conversion

   Rate : constant Float := 6.95; -- Prix d'un Dollar en Franc
```

```
   function Convert (F : Franc) return Dollar is
   begin
      return Dollar (Float (F) / Rate);
   end Convert;

   function Convert (D : Dollar) return Franc is
   begin
      return Franc (Float (D) * Rate);
   end Convert;

begin

   Account_BF := 10;
   Account_FR := 5;

   --  Transfert de compte

   Account_BF := Account_BF + Convert (Account_FR); -- c'est correct
   Account_FR := 0;

   --  Transfert dans l'autre sens

   Account_FR := Account_FR + Convert (Account_BF);
   Account_BF := 0;

end Bank;
```

[17] fichier : Genericite/Ada/V0/bank.adb

```
with Io;
procedure Bank is

   type Franc is new Integer;
   type Dollar is new Integer;

   --  Les types Franc et Dollar sont dérivés du type Integer
   --  Ils en pmossèdent toutes les caractèristiques
   --  Mais ils sont distincts

   Account_BF : Franc; -- le compte en Francs
   Account_FR : Dollar; -- le compte en Dollars

   --  On peut surcharger la fonction Convert qui sert à faire
   --  une vraie conversion ; ce sont les fonctions :
```

```
   function Convert (F : Franc) return Dollar;
   function Convert (D : Dollar) return Franc;

   --  Voici leur corps qui utilise la constante Rate
   --  comme taux de conversion

   Rate : Float := 6.95; -- Prix d'un Dollar en Franc

   function Convert (F : Franc) return Dollar is
   begin
      return Dollar (Float (F) / Rate);
   end Convert;

   function Convert (D : Dollar) return Franc is
   begin
      return Franc (Float (D) * Rate);
   end Convert;

begin

   Io.Put ("Donnez le montant en Francs : ");
   Io.Get (Account_BF);
   Io.Put ("Donnez le montant en Dollars : ");
   Io.Get (Account_FR);

   --  Transfert de compte

   Account_BF := Account_BF + Convert (Account_FR); -- c'est correct
   Account_FR := 0;

   Io.Put ("Voici le montant en Francs : ");
   Io.Put (Account_BF);
   Io.New_Line;
   Io.Put ("Voici le montant en Dollars : ");
   Io.Put (Account_FR);
   Io.New_Line;

   --  Transfert dans l'autre sens

   Account_FR := Account_FR + Convert (Account_BF);
   Account_BF := 0;

   Io.Put ("Voici le montant en Francs : ");
   Io.Put (Account_BF);
```

```
   Io.New_Line;
   Io.Put ("Voici le montant en Dollars : ");
   Io.Put (Account_FR);
   Io.New_Line;

end Bank;
```

[18] fichier : Genericite/Ada/V0/bank.log

```
$ gnatmake bank
gnatgcc -c bank.adb
bank.adb:38:06: invalid parameter list in call (use -gnatf for details)
bank.adb:40:06: invalid parameter list in call (use -gnatf for details)
bank.adb:48:06: invalid parameter list in call (use -gnatf for details)
bank.adb:48:06: possible missing instantiation of Text_IO.Integer_IO
bank.adb:51:06: invalid parameter list in call (use -gnatf for details)
bank.adb:51:06: possible missing instantiation of Text_IO.Integer_IO
bank.adb:60:06: invalid parameter list in call (use -gnatf for details)
bank.adb:60:06: possible missing instantiation of Text_IO.Integer_IO
bank.adb:63:06: invalid parameter list in call (use -gnatf for details)
bank.adb:63:06: possible missing instantiation of Text_IO.Integer_IO
gnatmake: "bank.adb" compilation error
```

[19] fichier : Genericite/Ada/V0/bank2.log

```
$ gnatmake -gnatf bank
gnatgcc -c -gnatf bank.adb
bank.adb:38:06: no candidate interpretations match the actuals:
bank.adb:38:12: expected type "Standard.Float"
bank.adb:38:12: found type "Franc" defined at line 4
bank.adb:38:12:   ==> in call to "Get" at io.ads:13
bank.adb:38:12:   ==> in call to "Get" at io.ads:10
bank.adb:38:12:   ==> in call to "Get" at io.ads:7
bank.adb:40:06: no candidate interpretations match the actuals:
bank.adb:40:12: expected type "Standard.Float"
bank.adb:40:12: found type "Dollar" defined at line 5
bank.adb:40:12:   ==> in call to "Get" at io.ads:13
bank.adb:40:12:   ==> in call to "Get" at io.ads:10
bank.adb:40:12:   ==> in call to "Get" at io.ads:7
bank.adb:48:06: no candidate interpretations match the actuals:
bank.adb:48:06: possible missing instantiation of Text_IO.Integer_IO
bank.adb:48:12: expected type "Standard.String"
```

```
bank.adb:48:12: found type "Franc" defined at line 4
bank.adb:48:12:   ==> in call to "Put" at io.ads:16
bank.adb:48:12:   ==> in call to "Put" at io.ads:14
bank.adb:48:12:   ==> in call to "Put" at io.ads:11
bank.adb:48:12:   ==> in call to "Put" at io.ads:8
bank.adb:51:06: no candidate interpretations match the actuals:
bank.adb:51:06: possible missing instantiation of Text_IO.Integer_IO
bank.adb:51:12: expected type "Standard.String"
bank.adb:51:12: found type "Dollar" defined at line 5
bank.adb:51:12:   ==> in call to "Put" at io.ads:16
bank.adb:51:12:   ==> in call to "Put" at io.ads:14
bank.adb:51:12:   ==> in call to "Put" at io.ads:11
bank.adb:51:12:   ==> in call to "Put" at io.ads:8
bank.adb:60:06: no candidate interpretations match the actuals:
bank.adb:60:06: possible missing instantiation of Text_IO.Integer_IO
bank.adb:60:12: expected type "Standard.String"
bank.adb:60:12: found type "Franc" defined at line 4
bank.adb:60:12:   ==> in call to "Put" at io.ads:16
bank.adb:60:12:   ==> in call to "Put" at io.ads:14
bank.adb:60:12:   ==> in call to "Put" at io.ads:11
bank.adb:60:12:   ==> in call to "Put" at io.ads:8
bank.adb:63:06: no candidate interpretations match the actuals:
bank.adb:63:06: possible missing instantiation of Text_IO.Integer_IO
bank.adb:63:12: expected type "Standard.String"
bank.adb:63:12: found type "Dollar" defined at line 5
bank.adb:63:12:   ==> in call to "Put" at io.ads:16
bank.adb:63:12:   ==> in call to "Put" at io.ads:14
bank.adb:63:12:   ==> in call to "Put" at io.ads:11
bank.adb:63:12:   ==> in call to "Put" at io.ads:8
gnatmake: "bank.adb" compilation error
```

[20] fichier : Genericite/Ada/V1/bank.adb

```
with Io;
with Ada.Text_IO;
procedure Bank is

   type Franc is new Integer;
   type Dollar is new Integer;

   --  Utilisation du paquetage générique Ada.Text_IO.Integer_IO
   --  pour obtenir les entrées/sorties sur les frans et les dollars

   package Franc_Io is new Ada.Text_IO.Integer_IO (Franc);
```

```
   package Dollar_Io is new Ada.Text_IO.Integer_IO (Dollar);

   Account_BF : Franc; -- le compte en Francs
   Account_FR : Dollar; -- le compte en Dollars

   function Convert (F : Franc) return Dollar;
   function Convert (D : Dollar) return Franc;

   Rate : Constant Float := 6.95; -- Prix d'un Dollar en Franc

   function Convert (F : Franc) return Dollar is
   begin
      return Dollar (Float (F) / Rate);
   end Convert;

   function Convert (D : Dollar) return Franc is
   begin
      return Franc (Float (D) * Rate);
   end Convert;

begin

   Io.Put ("Donnez le montant en Francs : ");
   Franc_Io.Get (Account_BF);
   Io.Put ("Donnez le montant en Dollars : ");
   Dollar_Io.Get (Account_FR);

   --  Transfert de compte

   Account_BF := Account_BF + Convert (Account_FR); -- c'est correct
   Account_FR := 0;

   Io.Put ("Voici le montant en Francs : ");
   Franc_Io.Put (Account_BF);
   Io.New_Line;
   Io.Put ("Voici le montant en Dollars : ");
   Dollar_Io.Put (Account_FR);
   Io.New_Line;

   --  Transfert dans l'autre sens

   Account_FR := Account_FR + Convert (Account_BF);
   Account_BF := 0;

   Io.Put ("Voici le montant en Francs : ");
```

```
   Franc_Io.Put (Account_BF);
   Io.New_Line;
   Io.Put ("Voici le montant en Dollars : ");
   Dollar_Io.Put (Account_FR);
   Io.New_Line;

end Bank;
```

[21] fichier : Genericite/Ada/V2/currencies.ads

```
package Currencies is

   type Franc is new Integer;
   type Dollar is new Integer;

   --  Fonctions de conversion :

   function Convert (F : Franc) return Dollar;
   function Convert (D : Dollar) return Franc;

   --  Procédures d'entrées/sorties :

   procedure Get (F : out Franc);
   procedure Get (D : out Dollar);
   procedure Put (F : in Franc);
   procedure Put (D : in Dollar);

end Currencies;
```

[22] fichier : Genericite/Ada/V2/currencies.adb

```
with Ada.Text_IO;
package body Currencies is

   --  Le corps des fonctions de conversions utilise la constante Rate
   --  comme taux de conversion

   Rate : constant Float := 6.95; -- Prix d'un Dollar en Franc

   -------------
   -- Convert --
   -------------
```

```
function Convert (F : Franc) return Dollar is
begin
   return Dollar (Float (F) / Rate);
end Convert;

-------------
-- Convert --
-------------

function Convert (D : Dollar) return Franc is
begin
   return Franc (Float (D) * Rate);
end Convert;

--  Utilisation du paquetage générique Ada.Text_IO.Integer_IO
--  pour obtenir les entrées/sorties sur les frans et les dollars

package Franc_Io is new Ada.Text_IO.Integer_IO (Franc);
package Dollar_Io is new Ada.Text_IO.Integer_IO (Dollar);

---------
-- Get --
---------

procedure Get (F : out Franc) is
begin
   Franc_Io.Get (F);
end Get;

---------
-- Get --
---------

procedure Get (D : out Dollar) is
begin
   Dollar_Io.Get (D);
end Get;

---------
-- Put --
---------

procedure Put (F : in Franc) is
begin
   Franc_Io.Put (F);
```

```
   end Put;

   ---------
   -- Put --
   ---------

   procedure Put (D : in Dollar) is
   begin
      Dollar_Io.Put (D);
   end Put;

end Currencies;
```

[23] fichier : Genericite/Ada/V2/bank.adb

```
with Currencies;
with Io;
procedure Bank is

   use Currencies;

   Account_BF : Franc; -- le compte en Francs
   Account_FR : Dollar; -- le compte en Dollars


begin

   Io.Put ("Donnez le montant en Francs : ");
   Get (Account_BF);
   Io.Put ("Donnez le montant en Dollars : ");
   Get (Account_FR);

   --  Transfert de compte

   Account_BF := Account_BF + Convert (Account_FR); -- c'est correct
   Account_FR := 0;

   Io.Put ("Voici le montant en Francs : ");
   Put (Account_BF);
   Io.New_Line;
   Io.Put ("Voici le montant en Dollars : ");
   Put (Account_FR);
   Io.New_Line;
```

```
   --  Transfert dans l'autre sens

   Account_FR := Account_FR + Convert (Account_BF);
   Account_BF := 0;

   Io.Put ("Voici le montant en Francs : ");
   Put (Account_BF);
   Io.New_Line;
   Io.Put ("Voici le montant en Dollars : ");
   Put (Account_FR);
   Io.New_Line;

end Bank;
```

[24] fichier : Genericite/Ada/V3/currencies.ads

```
package Currencies is

   type Franc is new Integer;
   type Dollar is new Integer;

   --  Fonctions de conversion :

   function Convert (F : Franc) return Dollar;
   function Convert (D : Dollar) return Franc;

end Currencies;
```

[25] fichier : Genericite/Ada/V3/currencies.adb

```
package body Currencies is

   --  Le corps des fonctions de convesrions utilise la constante Rate
   --  comme taux de conversion

   Rate : constant Float := 6.95; -- Prix d'un Dollar en Franc

   -------------
   -- Convert --
   -------------

   function Convert (F : Franc) return Dollar is
```

```
   begin
      return Dollar (Float (F) / Rate);
   end Convert;

   -------------
   -- Convert --
   -------------

   function Convert (D : Dollar) return Franc is
   begin
      return Franc (Float (D) * Rate);
   end Convert;


end Currencies;
```

[26] fichier : Genericite/Ada/V3/currencies-io.ads

```
package Currencies.Io is

   --  Les procédures d'entrées/sorties
   procedure Get (F : out Franc);
   procedure Get (D : out Dollar);
   procedure Put (F : in Franc);
   procedure Put (D : in Dollar);

end Currencies.Io;
```

[27] fichier : Genericite/Ada/V3/currencies-io.adb

```
with Ada.Text_IO;
package body Currencies.Io is

   --  Utilisation du paquetage générique Ada.Text_IO.Integer_IO
   --  pour obtenir les entrées/sorties sur les frans et les dollars

   package Franc_Io is new Ada.Text_IO.Integer_IO (Franc);
   package Dollar_Io is new Ada.Text_IO.Integer_IO (Dollar);

   ---------
   -- Get --
   ---------
```

```
   procedure Get (F : out Franc) is
   begin
      Franc_Io.Get (F);
   end Get;

   ---------
   -- Get --
   ---------

   procedure Get (D : out Dollar) is
   begin
      Dollar_Io.Get (D);
   end Get;

   ---------
   -- Put --
   ---------

   procedure Put (F : in Franc) is
   begin
      Franc_Io.Put (F);
   end Put;

   ---------
   -- Put --
   ---------

   procedure Put (D : in Dollar) is
   begin
      Dollar_Io.Put (D);
   end Put;

end Currencies.Io;
```

[28] fichier : Genericite/Ada/V3/bank.adb

```
with Currencies;
with Currencies.Io;
with Io;
procedure Bank is

   use Currencies;
```

```
   Account_BF : Franc; -- le compte en Francs
   Account_FR : Dollar; -- le compte en Dollars


begin

   Io.Put ("Donnez le montant en Francs : ");
   Currencies.Io.Get (Account_BF);
   Io.Put ("Donnez le montant en Dollars : ");
   Currencies.Io.Get (Account_FR);

   --  Transfert de compte

   Account_BF := Account_BF + Convert (Account_FR); -- c'est correct
   Account_FR := 0;

   Io.Put ("Voici le montant en Francs : ");
   Currencies.Io.Put (Account_BF);
   Io.New_Line;
   Io.Put ("Voici le montant en Dollars : ");
   Currencies.Io.Put (Account_FR);
   Io.New_Line;

   --  Transfert dans l'autre sens

   Account_FR := Account_FR + Convert (Account_BF);
   Account_BF := 0;

   Io.Put ("Voici le montant en Francs : ");
   Currencies.Io.Put (Account_BF);
   Io.New_Line;
   Io.Put ("Voici le montant en Dollars : ");
   Currencies.Io.Put (Account_FR);
   Io.New_Line;

end Bank;
```

[29] fichier : Exceptions/Ada/V1/sqrt.adb

```
function Sqrt (X : Integer) return Integer is
   R : Integer;
begin
   if X < 0 then
```

```
      raise Constraint_Error;
   end if;
   R := 0;
   while (R + 1) ** 2 <= X loop
      R := R + 1;
   end loop;
   return R;
end Sqrt;
```

[30] fichier : Exceptions/Ada/V1/roots.adb

```
with Sqrt, Io;
procedure Roots is
   I : Integer;
begin
   Io.Put ("Racines  de nombres divers");
   Io.New_Line (2);
   loop
      Io.Put ("Donnez un entier : ");
      Io.Get (I);
      exit when I = 0;
      Io.Put ("La racine de ");
      Io.Put (I);
      Io.Put (" est ");
      Io.Put (Sqrt (I));
      Io.New_Line;
   end loop;
   Io.New_Line;
   Io.Put ("Fin du programme");
end Roots;
```

[31] fichier : Exceptions/Ada/V2/sqrt.adb

```
function Sqrt (X : Natural) return Integer is
   R : Natural;
begin
   R := 0;
   while (R + 1) ** 2 <= X loop
      R := R + 1;
   end loop;
   return R;
end Sqrt;
```

[32] fichier : Exceptions/Ada/V2/roots.adb

```
with Sqrt, Io;
procedure Roots is
   I : Integer;
begin
   Io.Put ("Racines  de nombres divers");
   Io.New_Line (2);
   loop
      Io.Put ("Donnez un entier : ");
      Io.Get (I);
      exit when I = 0;
      Io.Put ("La racine de ");
      Io.Put (I);
      Io.Put (" est ");
      begin
         Io.Put (Sqrt (I));
      exception
         when Constraint_Error => Io.Put ("non calculable");
      end;
      Io.New_Line;
   end loop;
   Io.New_Line;
   Io.Put ("Fin du programme");
end Roots;
```