Attributs: Variables, Types de base

5

Les attributs représentent les caractéristiques de la classe.

1. Variables

1.1. Déclaration et affectation

La première chose pour utiliser une variable est de la déclarer. On peut la déclarer soit à l'intérieur de la classe soit à l'intérieur d'une routine. Pour la déclaration, la syntaxe est la suivante:

```
une_variable : TYPE
```

La variable *une_variable* par convention est toujours écrite en minuscule suivi de deux points « : » et terminée par le nom de la classe (ou type).

Si on veut affecter (assigner) une valeur à cette variable, on utilise l'opérateur d'affectation « := ».

```
un_entier : INTEGER -- Déclaration
un_entier := 5
```

On assigne la valeur entière 5 à la variable *un_entier*.

3.1.2. Visibilité

Les variables définies dans une classe appartiennent à deux catégories: les variables d'instance et les variables locales

Variables d'instance

Les variables d'instance sont déclarées dans la classe au même titre que les routines. Elles sont accessibles et visibles de n'importe quel endroit à l'intérieur de la classe. Par contre, il est impossible de manipuler cette variable à l'extérieur de la classe.

Variables locales

Une variable locale ne peut être utilisée qu'à l'intérieur d'une routine. Elle est inaccessible pour les autres routines. Elle doit donc être déclarée dans une routine dans la rubrique **local** placée juste avant le commencement du bloc de code **do....end.**

La classe A ci-dessous possède une variable d'instance i et une variable locale dans la routine *subtract*. La variable i peut être appelée n'importe où dans la classe alors que j est limité au bloc de

code délimité par *subtract*.

```
class A

feature -- Variable d'instance
    i : INTEGER

feature -- Routine
    subtract(increment : INTEGER) is
    local
          j : INTEGER
    do
          i := i - increment -- Correct; i est utilisable
          ....
    end


end -- end of class A
```

Les deux routines suivantes sont incorrectes.

```
    add( increment : INTEGER) is
    do
          j := j + increment -- Incorrect!!; j est inconnu .
    end

    multiply (i : INTEGER) is
          -- ERREUR!!, l'argument i est en conflit avec la variable i.
    do
          i := i * i
    end
```

La routine *add* est incorrecte car elle fait appel à la variable locale j inaccessible dans ce bloc de code. Enfin, la routine *multiply* présente un conflit de noms entre la variable d'instance i et l'argument de *multiply* (qui est une variable locale de la routine multiply) (lui aussi nommé j)

Utilisation par un objet

Il est impossible de modifier un attribut de l'extérieur de la classe, puisque ce sont des variables d'instance.

```
un_objet : A
......
un_objet.i := 45 -- Faux!!
```

Il est nécessaire de définir des routines nommées "accesseurs" pour accéder aux attributs d'une classe. Les noms de ces routines utilisent les préfixes set_ et get_ suivies du nom de la variable d'instance. Par exemple, pour une variable *width*, on définira deux routines: *set_width* et get_*width*.

Les lignes de code suivantes présentent deux accesseurs set_value et get_value

```
class A
feature {ANY}
set_value (value : INTEGER) is
```

```
    do
        i := value
    end

get_value : INTEGER is
    do
        Result := i
    end
end -- end of class A

class TEST
.....
feature {ANY}
    un_objet : A
......
    make is
        do
            un_objet.set_value(a_value) -- Affectation
            io.put_integer(un_objet.get_value) -- Lecture
        end
.......
```

Toutefois, en Eiffel, même si la variable est inaccessible en écriture, il est possible de la lire sans passer par un accesseur. Par exemple, la variable i de la classe A peut être utilisé de la façon suivante:

```
un_objet : A
......
io.put_integer(un_objet.i) — Correct
```

Tout ceci est valable car i est déclaré dans une clause publique feature {ANY}.

2. Constantes

Une constante a une déclaration et une typographie particulières:

```
Une_constante : INTEGER is 4
Autre_constante : DOUBLE is 5.43
Encore_une_constante : CHARACTER is 'Z'
Encore_une_autre_constante : STRING is "Je suis une constante"
Premier,Deuxieme,Troisieme: INTEGER is unique -- Enumération
```

Une constante est définie comme une variable à deux différences près. (i) Le nom de la variable commence par une majuscule. (ii) Après la déclaration, une valeur est affectée à la variable via le mot clé **is** suivie de la valeur de la variable. Il est impossible de modifier le contenu de la variable pendant l'exécution de l'objet.

3. Info ou intox, vrai ou Faux: BOOLEAN

Le BOOLEAN peut prendre deux valeurs: **True** (vrai) et **False** (faux). Toutes les opérations de comparaison renvoient un BOOLEAN comme réussite (ou échec) du test de comparaison. Par exemple,

```
    17 >  18 -- False
    17 >= 17 --True
```

Les BOOLEANs sont très pratiques lorsque vous utilisez des « drapeaux » (« flags » en anglais) pour voir si un objet possède une propriété. Par exemple, une classe ACIDE_AMINE pourrait comprendre une fonction *is_aromatic* qui renvoie **True** lorsque l'objet est Phe, Trp ou Tyr et **False** dans les autres cas.

```
is_aromatic : BOOLEAN is
do
    -- Faire l'implémentation
end
```

4. Des chiffres ....

Les nombres sous Eiffel sont des objets définis par des classes comme tout ce qui est utilisé dans ce langage. Tous les nombres dérivent d'un ancêtre commun NUMERIC. On distingue deux grandes familles de nombres: les nombres entiers de classe INTEGER et les nombres réels (dits à virgule flottante) de classe REAL. Chacune de ces familles est elle-même subdivisée en sous-classes en fonction de la précision souhaitée ce qui correspond en fait au nombre de bits (8, 16, 32 ou 64 bits) nécessaires pour stocker ces nombres.

4.1. INTEGER

Ce type permet de définir des nombres entiers stockés sur 32 bits (depuis la version 2.1) et donc pouvant prendre des valeurs comprises entre $2^{31} – 1$ et $-(2^{31} -1)$. Les principales opérations arithmétiques pour les INTEGER sont:

+	signe « plus » . Exemple: +65
-	signe « moins ». Exemple: -67
^	puissance. Exemple 3^2 égal 9
*	multiplication. Exemple 3 * 2 égal 6
/	division. Exemple: 3/ 2 égal 1.5 (attention, le résultat est un REAL)
+	addition. Exemple: 3+2 égal 5
-	soustraction. Exemple: 3 - 2 égal 1
//	quotient de la division euclidienne . Exemple: 7 // 2 égal 3
\\	modulo. Exemple: 7 \\ 2 égal 1 (reste 1)

Remarque: Toutes les opérations des INTEGER sont décrites dans l'interface de la classe INTEGER à l'adresse http://smarteiffel.loria.fr/libraries/index.html

4.2. INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_64

Il est possible de travailler avec des variables numériques qui stockent les nombres avec 8 bits (INTEGER_8), 16 bits (INTEGER_16), 32 bits (INTEGER_32) et 64 bits (INTEGER_64). Pour les calculs courants, la classe INTEGER suffit à elle-même, mais dans certains types d'application comme le traitement d'images, la gestion de la mémoire est cruciale et les données doivent prendre le moins de place possible; on cherche donc à stocker les nombres dans le plus petit espace mémoire (INTEGER_8, INTEGER_16). Au contraire, si on travaille sur des grands nombres, on prendra des INTEGER_64.

En fonction du nombre de bits réservé pour la variable numérique, le nombre ne doit pas prendre plus d'espace mémoire que ce qui a été réservé initialement, il existe des bornes inférieure et supérieure:

Classes	*inférieure*	*supérieure*
INTEGER_8	-128	127
INTEGER_16	-32768	32767
INTEGER_32 ou INTEGER	-2147483648	2147483647
INTEGER_64	-92233720368547808	9223372036854775807

Ces bornes sont accessibles dans la classe PLATFORM qui est un ancêtre commun à toutes les classes. On peut donc afficher toutes les bornes de la manière suivante:

```
class LIMITS
creation
    make
feature
    make is
        do
            io.put_string("Minimum : " + Minimum_integer_8.out + "%N")
            io.put_string("Maximum : " + Maximum_integer_8.out + "%N")
            io.put_string("Minimum : " + Minimum_integer_16.out + "%N")
            io.put_string("Maximum : " + Maximum_integer_16.out + "%N")
        end
    end
end -- end of class LIMITS
```

3.4.2. NATURAL

Les objets NATURAL – récemment apparus dans Eiffel – permettent de stocker des nombres positifs (dits non signés). On retrouve les mêmes types de classes que pour les INTEGER avec NATURAL, NATURAL_8, NATURAL_16, NATURAL_32 et NATURAL_64. Un objet de type NATURAL_8 stockerait un nombre compris entre [0;255]

```
io.put_string("Minimum : " + Minimum_natural_8.out + "%N") --    0
io.put_string("Maximum : " + Maximum_natural_8.out + "%N") --  255
```

Ces classes ne sont malheureusement pas pour *l'instant* implantées dans SmartEiffel.

3.4.3.REAL

Le REAL correspond à un nombre en *virgule* flottante stocké sur 64 bits. Les opérations arithmétiques sont les mêmes que celles utilisées par les INTEGER. On trouvera en plus les fonctions mathématiques classiques (trigonométriques, logarithmiques,etc.). Ces routines doivent être utilisées en les plaçant **après** l'objet comme n'importe quel message à appliquer à un objet.

Exemples:

```
r : REAL
....
r := 16.0
r.sqrt          -- Calcul de la racine carrée de r soit 4.0.
(45.0 / Deg).cos      -- cos(45.0 * Pi / 180.0)
(60.0 / Deg).sqrt     -- est aussi valable
(60.0 / Deg).cos.sqrt -- sqrt(cos(60.0 * Pi / 180.0) )
```

Pour la dernière ligne, le message *sqrt* (*square root* = racine carrée) est envoyé à l'objet créé lors de l'exécution du message *cos* (cosinus) envoyé à l'objet (60.0 / Deg) de la classe REAL. Au final, l'opération 60.0 / Deg retourne un objet REAL qui correspond au nombre 1.047198 (Il s'agit en fait de convertir 60.0° en radians). Puis, à ce nombre est appliqué la routine *cos* qui créé à son tour un nouvel objet REAL de valeur 0.5. Et enfin, le message sqrt est envoyé à 0.5 pour donner l'objet final de valeur 0.707.

3.4.3.Typage et conversion

Le langage Eiffel est très strict en ce qui concerne le type de l'objet. Par exemple, la ligne suivante

```
u, v , w: INTEGER

w := u / v -- FAUX!!
```

La division de deux entiers u par v retourne un résultat de type REAL qu'il est impossible d'affecter à w (une variable INTEGER). Pour pouvoir effectivement mettre le résultat de la division dans w, il faut arrondir ou tronquer le résultat pour le convertir en INTEGER.

```
u, v , w: INTEGER

w := (u / v).rounded -- Résultat arrondi.
```

Autre exemple avec le calcul de la racine carrée d'un nombre réel

```
u, v: REAL

v := u.sqrt.rounded -- FAUX!!
v := u.sqrt.to_real -- Correct. Résultat DOUBLE converti en REAL.
```

Il existe de nombreuses routines de conversion entre INTEGER, REAL et DOUBLE. Heureusement pour le programmeur « client », elles sont toutes de la forme *to_<nom_du_type>*. Par exemple, to_real, to_double,...

3.4.4.Les nombres et les variables numériques

En Eiffel, il n'est fait aucune différence entre les nombres exprimés dans un système de numération (base 10 (décimal), 16 (hexadécimal), 8 (octal) ou 2(binaire)) et les variables qui sont des "conteneurs" de nombres (INTEGER, NATURAL, REAL). Il faut donc un mécanisme particulier pour savoir si un nombre comme « 1 » est un INTEGER_8, NATURAL_16 ou REAL_32.

Par défaut, SmartEiffel applique la règle suivante. Si un nombre peut être stocké dans la variable occupant le moins d'octets possible, alors ce nombre prend le type de la variable.

Par exemple, 2 peut être stocké dans une variable de type INTEGER_8 (un entier stocké sur 8 bits), alors 2 est de type INTEGER_8. De la même façon, 127 est de type INTEGER_8 car il est compris entre [-128; 127]; au contraire, 128 ne peut plus être contenu dans une variable de type INTEGER_8, donc 128 sera de type INTEGER_16 (permettant le stockage de nombres compris entre [-32768; 32767]. Le programme suivant montre la classe de divers nombres en fonction de leur valeur:

```
io.put_string(2.generator + "%N")          -- display INTEGER_8
io.put_string(127.generator + "%N")        -- display INTEGER_8
io.put_string(128.generator + "%N")        -- display INTEGER_16
io.put_string((2.0).generator + "%N")      -- display REAL
```

Toutefois, ce système automatique présente de nombreuses limitations car les lignes suivantes sont incorrectes.

```
i : REAL_32
...
i := 1.0 -- Faux car 1.0 est REAL_64 qui est différent de REAL_32
```

Pour combler les limitations du système précédent, SmartEiffel dispose d'une écriture spécifique dite "manifeste" pour forcer un nombre à prendre un type particulier. La syntaxe est la suivante:

```
{<TYPE> <value>}
```

On place le nombre entre parenthèses précédé de la classe.

Par exemple,

```
i : INTEGER_16
j : REAL_64
k : INTEGER_32
.....
i := {INTEGER_16 127}
j := {REAL_64 45}
k := {INTEGER_32 -10}
....
```

Ceci est valable aussi pour des routines dont les arguments sont par exemple, des REAL_32

```
gl_color_3f(red, green, blue : REAL_32) is
.....
gl_color_3f({REAL_32 1.0}, {REAL_32 1.0}, {REAL_32 1.0})
```

Les autres nombres: NUMBER

< A FAIRE>

5. ....et des lettres

5.1. CHARACTER

Les caractères en Eiffel sont délimités par des apostrophes. Par exemple,

```
un_caractere : CHARACTER
......
un_caractere := 'a'
un_caractere := 'A'
un_caractere := '@'
....
un_caractere := '%/27/'  -- Code ASCII 27 correspondant à Escape
```

Truc et astuces: Pour saisir le caractère par son code ASCII, taper %/n/ où n est le code ASCII. Il existe de plus des caractères spéciaux pour les fins de ligne, les tabulations comme '%N', '%T'.

5.2. STRING et UNICODE_STRING

voir chapitre sur les collections

6. Application: classe EBO_ATOM

Dans un fichier Protein DataBank (PDB), un atome est repéré par les caractéristiques suivantes:

ses coordonnées atomiques (x, y , z)

son nom

son symbole chimique

occupancy

Les variables d'instance

```
class EBO_ATOM
feature
    x : REAL
          -- X coordinate

    y : REAL
          -- Y coordinate

    z : REAL
          -- Z coordinate

    name : STRING

    symbol : STRING

    occupancy : REAL

end -- end of class EBO_ATOM
```