Ecrire une classe
Utiliser des objets

3

1. Définition

Lorsque dans la vie quotidienne, vous utilisez des objets manufacturés, ceux-ci ont été fabriqués à partir de plans. Un plan permet de construire (créer) une multitude d'objets parfaitement identiques. Dans un langage de classes, on retrouve la même démarche. La **classe** est le « plan » permettant de créer plusieurs objets (**instances**). La classe contient donc une description de la **structure** et du **comportement** de l'objet.

En Eiffel, une classe est définie par des attributs et des routines. Les **attributs** caractérisent l'état d'une instance de la classe alors que les **routines** (procédures ou fonctions) vont manipuler les instances de la classe. Le programmeur va donc définir de nouvelles classes qui serviront à créer des objets lors de l'exécution du programme. Il est possible de créer autant d'objets qu'on le désire et ensuite, de manipuler ces objets comme des éléments existant dans le problème à résoudre.

1.1. Ma première classe

A l'aide de votre éditeur de texte favori, créez un nouveau fichier « bonjour.e » (nom en minuscules avec l'extension « .e »), puis saisissez le texte suivant.

```
class BONJOUR

creation
    make

feature
    make is
    do
          io.put_string(« Bonjour chez vous!%N »)
    end

end -- end of class BONJOUR
```

Après avoir sauvegardé votre texte, lancez la compilation

```
se c bonjour.e -o bonjour -clean
```

puis le fichier exécutable nouvellement créé

```
./bonjour
```

Si tout s'est correctement déroulé, vous devez lire à l'écran, la phrase

```
Bonjour chez vous
```

1.2. Syntaxe

Une classe est définie par le mot clé **class** et est terminée par le mot clé **end** suivi d'un commentaire dans lequel on doit trouver la phrase -- *end of class [nom_de_la_classe].* Entre ces deux bornes **class .... end**, on trouve la définition de la classe qui est subdivisée en deux parties définies par les clauses **creation** et **feature**. La syntaxe est la suivante:

```
class <NOM_DE_LA_CLASSE>

creation

feature

end -- end of class <NOM_DE_LA_CLASSE >
```

1.3. Les conventions de définition de classes en Eiffel

- ✔ Un fichier texte ne contient qu'une classe.
- ✔ Le nom de la classe est toujours en majuscule.
- ✔ Le nom du fichier texte doit être identique au nom de la classe mais en minuscules et avec l'extension « .e ». Par exemple, la classe MA_SEQUENCE doit être saisie dans le fichier 'ma_sequence.e'
- ✔ Les caractères accentués ne sont pas autorisés dans le nom des fichiers et des classes.

2. La rubrique creation

Elle indique quel(s) est(sont) le(s) constructeur(s) de la classe, c'est à dire les routines associées lors de la création de l'objet. Un constructeur est une routine qui sert à l'initialisation de l'objet lors de sa création. Dans la classe BONJOUR, il existe un constructeur *make* qui est automatiquement reconnu par le compilateur comme étant la routine principale (équivalent de main() en langage C) à exécuter en premier.

Remarque: Rien n'empêche d'utiliser un autre nom de constructeur mais il faudra le spécifier au compilateur. Par exemple, si on remplace *make* par *mon_bonjour* dans le fichier bonjour.e, il faudra taper la commande suivante:

```
se c bonjour.e mon_bonjour -o bonjour -clean
```

3. La rubrique feature

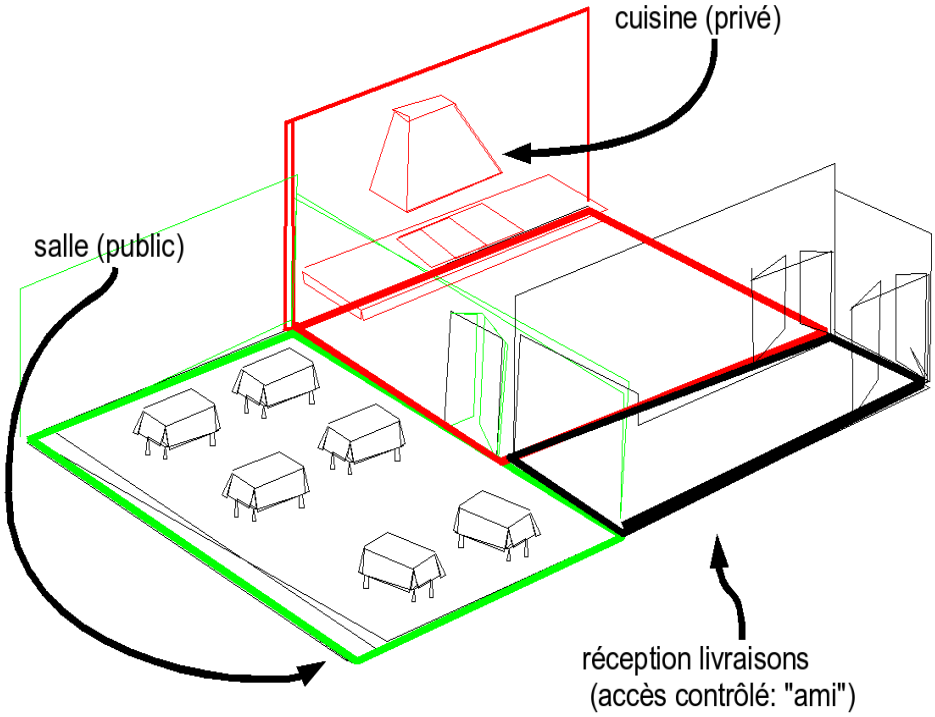
Elle contient la définition des caractéristiques et du comportement de la classe.

3.1. La classe ? un restaurant !!

Une classe est organisée comme un restaurant dont le schéma est présenté dans la figure xxx. On y distingue plusieurs zones d'activité qui ne s'adressent pas aux mêmes catégories de personnes. La salle de restaurant est publique. Les clients sont autorisés à y circuler et à "utiliser" les divers services proposés par le restaurant c'est à dire ici à consommer les plats. Dans cette zone publique,

les clients ne sont pas autorisés à modifier ces services (le client ne peut pas faire la cuisine). Au contraire, la cuisine est une zone privée où on confectionne les divers plats; ici les cuisiniers sont habilités à modifier les recettes. Enfin, il existe une ou plusieurs zones réservées à une certaine catégorie de personnels extérieur au restaurant (les livreurs, par exemple): c'est une zone amie où seuls les amis sont autorisés à y pénétrer.

Figure xxx: Organisation d'une classe similaire à celle d'un restaurant.



Les trois zones d'activité d'un restaurant sont:

(i) La salle publique: on y trouve des clients et le personnel du restaurant. Les clients consomment les divers plats sans savoir comment ceux-ci sont confectionnés.

(ii) La cuisine privée: seul le personnel du restaurant est autorisé à y circuler. Cet endroit est stratégique puisque les différents plats y sont fabriqués.
(iii) La zone de livraisons ni publique ni privée: elle a un accès contrôlé où du personnel extérieur au restaurant est autorisé à y circuler.

Visibilité des feature

Dans une classe, on retrouve ces trois zones:

(i) la zone publique est réservée au programmeur "client"; celui-ci va utiliser des fonctionnalités d'une classe programmée par une autre personne. Il ne se soucie pas de l'implantation: il est un utilisateur.

(ii)la zone privée contient l'implantation des fonctionnalités de la classe. Seul le programmeur "propriétaire" de la classe est autorisé à la modifier.

(iii)la zone "amie" est uniquement accessible à certaines classes. C'est la zone la plus difficile à circonscrire.

En pratique, la rubrique **feature** est répétée plusieurs fois dans la classe pour délimiter les zones

décrites ci-dessus. La syntaxe consiste à faire suivre le mot clé **feature** du nom de la (ou des) classe(s) dont l'accès est autorisé, entre accolades comme ci-dessous:

```
feature {FOO}
```

La zone publique appelée aussi **interface** est définie en y accolant la classe ANY (l'interface est destinée au programmeur « client »)

```
feature {ANY}
    -- Ici, les routines sont accessibles à tout le monde (public)
```

La zone privée est réservée à un usage interne (la « cuisine » où sont regroupés les attributs et les routines destinées à l'implémentation de la classe).

```
feature {NONE}
    -- Sous cette section, tout est privé
    -- Seule l'instance de cette classe peut utiliser cette partie.
```

La zone amie

```
feature {STRING; CHARACTER}
    -- Nous sommes dans un cas intermédiaire: ni public ni privé
    -- Politique « amie »; seuls les « amis » sont autorisés.
    -- Ici, les routines sont uniquement accessibles
    -- aux STRING et CHARACTER.
```

Variabilité de syntaxe:

On trouve selon les programmeurs, deux styles de définition des **feature** publiques et privées.

Par défaut, le mot clé **feature** seul équivaut à **feature {ANY}**

```
feature
    -- Politique par défaut. Tout est public.
```

De même, **feature {}** équivaut à **feature {NONE}**

```
feature {}
    -- Sous cette section, tout est privé
```

Pour la zone publique, je préconise de toujours utiliser **feature {ANY}** pour éviter toute confusion. De même, on suivra l'écriture préconisée par l'équipe de SmartEiffel pour les zones privées c'est à dire en utilisant f**eature {}**; bien qu'à mon avis, on peut confondre **feature** et **feature {}**. Le choix fait par SmartEiffel est justifié par le fait que NONE n'est pas une classe mais simplement un mot clé.

Conseil 1: Pour vos premières classes, essayez de trier les **feature** en deux blocs: (i) routines publiques et (ii) privées. Les cas intermédiaires "amis" sont plus difficiles à gérer.

Conseil 2 : Il est conseillé de subdiviser les **feature** de même visibilité en sous-blocs de manière à regrouper leur contenu dans la classe (par exemple, la zone publique peut être subdivisée en trois **feature{ANY}** pour classer (i) les constructeurs, (ii) les attributs publics, et (iii) les méthodes publiques d'accessibilité).

3.2. Que contiennent les "feature"?

Dans les rubriques **feature**, on va définir deux types de choses:

Les **attributs** qui sont les caractéristiques de la classe (par exemple, les attributs de la classe RESTAURANT seraient four, chaise, serveur, cuisinier,etc.).

Les **routines** ou **méthodes** qui sont des blocs de code permettant de définir un comportement à notre classe (par exemple, pour la classe RESTAURANT, on aurait *cook_foie_gras*);

3.3.Afficher la partie publique d'une classe

Pour le programmeur « client », il est intéressant de lire des renseignements sur une classe donnée. Dans SmartEiffel, il existe un utilitaire **short** qui permet d'extraire la documentation du fichier source d'une classe. Ce programme récupère comme information :

- ✔ Les commentaires placées dans l'entête de la classe
- ✔ Toutes les routines et attributs publics
- ✔ Tous les commentaires placées après les routines /attributs publics

Modifiez le fichier bonjour.e de manière à rajouter les commentaires. Un commentaire est une ligne commençant par deux tirets « -- ».

```
class BONJOUR

creation
    make

feature
    make is
          -- make est le constructeur de la classe BONJOUR
          -- Il affiche la phrase Bonjour chez vous!
    do
          io.put_string("Bonjour chez vous!%N")
    end

    une_variable : INTEGER
          -- Une variable pour l'exemple

feature {NONE} -- Privé
    une_variable_privee : REAL
          -- « Arriere cuisine »

end -- end of class BONJOUR
```

Après avoir sauvegardé le fichier bonjour.e, lancez:

```
se short -html2 bonjour.e > bonjour.html
```

Visualisez le fichier bonjour.html avec votre navigateur web préféré.

```
class interface BONJOUR

creation
   make
      -- make est le constructeur de la classe BONJOUR
      -- Il affiche la phrase Bonjour chez vous!
```

```
feature(s) from BONJOUR
   make
      -- make est le constructeur de la classe BONJOUR
      -- Il affiche la phrase Bonjour chez vous!

   une_variable: INTEGER
      -- Une variable pour l'exemple

end of BONJOUR
```

Uniquement la partie publique destinée au programmeur « client » (l'interface) est retranscrite dans ce fichier. L'utilisation de **short** est très utile pour vous rendre compte de ce que vous avez laissé public dans une classe (surtout quand cette classe hérite de plusieurs parents).

Remarque: Sur le site web de SmartEiffel, toutes les interfaces des classes définies par l'équipe de SmartEiffel sont consultables dans le sous-menu 'Libraries doc.'.

4. Conventions typographiques utilisées par Eiffel

Cette section est directement inspirée des conventions utilisées dans le projet Gobo (Gobo Eiffel de Eric Bezault à l'adresse www.gobosoft.com) qui sont elles-mêmes tirées des spécifications officielles publiées dans *Object-Oriented Software Construction, Second Edition* (appelé communément *OOSC-2*)

4.1. Noms des classes

Les noms des classes sont impérativement en majuscules.

Eiffel n'a pas de système d'espaces de noms comme dans d'autres langages comme C++, ce qui signifie qu'une classe ATOM dans le dossier /path/ebo/kernel peut être confondue avec une classe ATOM située dans /path/everywhere/kernel. Pour éviter tout problème de confusion, il est d'usage d'ajouter au début du nom de la classe, le nom (ou un acronyme ou surnom) de la librairie. Par exemple, pour EBO, on définira la classe EBO_ATOM, EBO_CHAIN, EBO_HELIX, etc.

Conseil: Evitez des préfixes trop longs (un préfixe de 4 caractères est considéré comme le maximum).

4.2. Noms des attributs et routines

Les noms sont en minuscules et si ceux-ci sont composés de différents mots, ils doivent être séparés par des caractères soulignés ("underscore"). Exemple:

```
get_length
from_collection
```

Il est d'usage d'éviter les abréviations à moins que celles-ci soient communément admises dans la communauté travaillant dans le domaine scientifique visé. Exemple, pour un bioinformaticien:

```
blast
get_fasta
```

seront bien évidemment des acronymes acceptés.

Les constantes sont des attributs particuliers qui sont écrits en minuscules à l'exception du 1er caractère en majuscule. Exemple, dans la classe MATH_CONSTANTS de SmartEiffel, on trouve:

```
Pi
Deg
```

Souvent, les noms des routines retournant un BOOLEAN sont construits comme des interrogations. Exemple:

```
is_valid
is_empty
```

Les noms des constructeurs peuvent être construits à partir de 'make', 'from' ou 'with'.

4.3. Indentations

Afin que le code soit lisible par tous, il est d'usage d'utiliser des indentations pour améliorer la clarté du code. Ces indentations sont réalisées avec des tabulations (touche 'tab' sur le clavier). La touche 'espace' est à proscrire et surtout le mélange de caractères 'espace' et tabulations qui peuvent rendre difficile la lecture d'un éditeur de texte à un autre. Le mieux est de choisir un éditeur de texte qui visualise les 'tab'.

L'exemple ci-dessous (extrait des conventions de GOBO) montre la disposition recommandée pour une routine. Ici, les tabulations sont représentées par le caractère 'souligné' (underscore) afin de visualiser l'utilisation de la touche 'tab':

```
_   set_foo (a_foo: like foo) is
_   _   _   -- Set `foo' to `a_foo'.
_   _   require
_   _   _   a_foo_not_void: a_foo /= Void
_   _   do
_   _   _   foo := a_foo
_   _   ensure
_   _   _   foo_set: foo = a_foo
_   _   end
```

Voici un exemple plus complet des conventions d'indentation pour une classe (extrait du chapitre des conventions typographiques de GOBO):

```
indexing

_   description:

_   _   "Short description of the class"

_   library:    "EBO: Eiffel Bioinformatics OpenGL"
_   author:     "Jean Dupont"
_   copyright:  "Copyright (c) 2005, Jean-Christophe Taveau and others"
_   license:    "GPL: General Public License"
_   date:       "$Date: 2003/02/07 11:57:51 $"
_   revision:   "$Revision: 1.2 $"

class BAR [G -> TOTO]

inherit
```

```
_   BAZ
_   _   rename
_   _   _   oof as foo,
_   _   _   f as g
_   _   redefine
_   _   _   foo, bar
_   _   end

creation

_   make, make_from_string

feature {NONE} -- Initialization

_   make (a_foo: FOO) is
_   _   _   -- Create a new bar.
_   _   require
_   _   _   a_foo_not_void: a_foo /= Void
_   _   do
_   _   _   set_foo (a_foo)
_   _   ensure
_   _   _   foo_set: foo = a_foo
_   _   end

_   make_from_string (a_string: STRING) is
_   _   _   -- Create a new bar from `a_string'.
_   _   require
_   _   _   a_string_not_void: a_string /= Void
_   _   do
_   _   _   !! foo.make_from_string (a_string)
_   _   end

feature -- Access

_   foo: FOO
_   _   _   -- Foo

feature -- Setting

_   set_foo (a_foo: like foo) is
_   _   _   -- Set `foo' to `a_foo'.
_   _   require
_   _   _   a_foo_not_void: a_foo /= Void
_   _   do
_   _   _   foo := a_foo
_   _   ensure
_   _   _   foo_set: foo = a_foo
_   _   end

invariant

_   foo_not_void: foo /= Void

end -- class BAR
```

4.4. Clause Indexing

Bien que cette clause ne soit pas utilisée par l'équipe de SmartEiffel. Dans le cours, on placera cet en-tête en début de chaque classe nouvellement créée:

```
indexing

_   description:

_   _   "Short description of the class"

_   library:    "EBO: Eiffel Bioinformatics OpenGL"
_   author:     "Jean Dupont"
_   copyright:  "Copyright (c) 2005, Jean-Christophe Taveau and others"
_   license:    "GPL: General Public License"
_   date:       "$Date$"
_   revision:   "$Revision"
```

Les champs 'date' and 'revision' sont automatiquement mis à jour par CVS.

4.5. Commentaires

Les commentaires en Eiffel sont définies par un double tiret '--' en début de chaque ligne. Si vous souhaitez faire un commentaire sur plusieurs lignes vous devez répéter le double tiret à chaque fois. Par exemple:

```
-- A comment
-- line #1
-- line #2
-- line #3
```

On ajoute un commentaire court (souvent un seul mot) après une clause **feature** comme par exemple:

```
feature -- Access
feature {NONE} -- Initialization
```

Généralement, on place un commentaire après chaque attribut et routine de la manière suivante:

```
feature -- Access

_   title: STRING
_   _   _   -- Title displayed in the title bar

feature -- Setting

_   set_title (a_title: like title) is
_   _   _   -- Set `title' to `a_title'.
_   _   require
_   _   _   a_title_not_void: a_title /= Void
_   _   do
_   _   _   title := a_title
_   _   ensure
_   _   _   title_set: title = a_title
```

```
_   _   end
```

On trouve aussi selon les auteurs des commentaires en fin de classe (après le end final) relatifs à la licence ou une description d'un problème concernant la classe.

4.6. Les points virgules ("semicolon")

Les points virgules sont optionnels en Eiffel et le compilateur SmartEiffel arrive très bien à s'en sortir sans point virgule. On ne les utilise que dans deux cas:

(i) pour séparer plusieurs arguments d'une routine:

```
        f (a_foo: FOO; a_bar: BAR) is
```

(ii) pour séparer plusieurs instructions placées sur la même ligne (bien que cette utilisation doit être limitée). Il est mieux d'avoir une instruction par ligne:

```
        print ("Hello "); print (you.name)
```

(iii) pour lever certaines ambiguïtés (rares, il faut bien le dire):

```
        foo.bar;
        (baz).do_something
```

Dans l'exemple ci-dessus, le compilateur peut confondre avec:

```
foo.bar(baz).do_something
```

4.7. Assertions

Il est préférable de joindre des assertions (pré- et post-conditions) aux routines ainsi qu'une clause 'invariant' aux classes pour faciliter le débogage.

Il est d'autre part recommandé pour localiser facilement l'assertion mise en cause au cours du débogage, d'associer un message sous forme d'une phrase courte à l'assertion comme le montre l'exemple ci-dessous

```
_   foo_not_void: foo /= Void
```

Lors de l'exécution, le message 'foo_not_void' s'affichera si cette clause n'est pas respectée.

5. L'Objet

Une fois la classe écrite, il faut passer à la création de l'objet, ce qui se fait en deux étapes:

5.1. Déclaration

Dans un langage compilé, il est obligatoire de déclarer l'objet c'est à dire d'indiquer à quelle classe appartient la variable. Ainsi, quand on déclare la chaîne de caractères str:

```
      str : STRING
```

on annonce que 'str' est de type STRING.

5.2. Création de l'objet

La création d'un objet s'appelle une instanciation qui consiste à créer une instance (exemplaire) à partir de la classe. Cette instanciation se fait au moyen du mot clé **create**

```
une_string : STRING
create une_string.make
```

Le constructeur peut contenir des arguments; par contre, il ne doit jamais retourner de valeur. Par exemple:

```
create a_point.with(1.0, 2.3, 5.46) -- correct
i := create a_point.make_default -- not correct
```

Cette opération d'instanciation consiste à réserver de la place en mémoire pour l'objet comme le montre la figure xxx pour la création de 'atom' de classe ATOM.

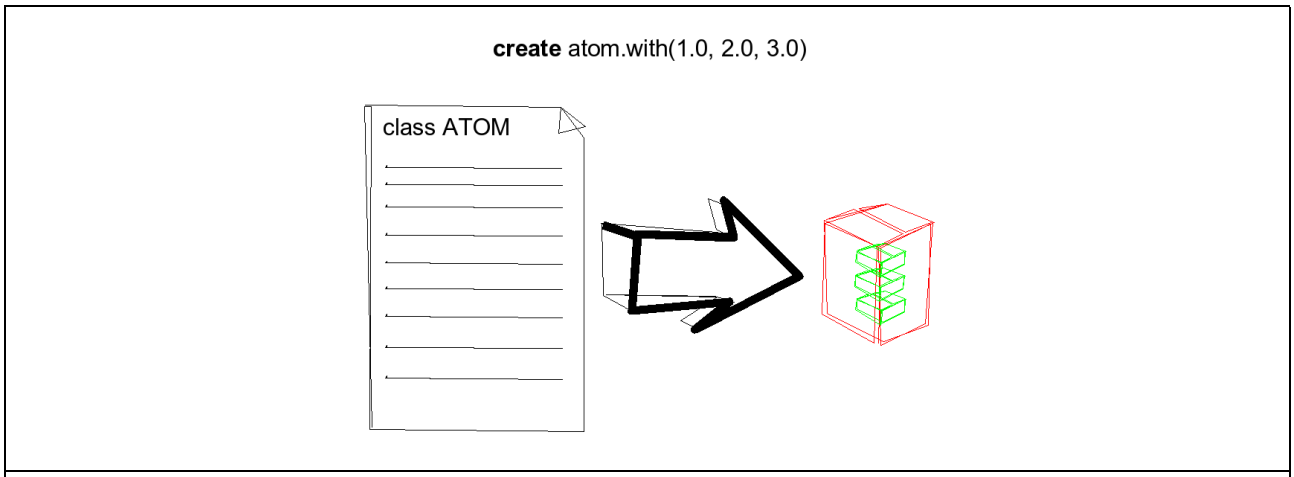


Figure xxx: Création d'un objet 'atom' de classe ATOM. Ici, le constructeur est 'with' et attend trois arguments.

5.3.Manipulation d'objets

Chaque langage de programmation a sa façon de manipuler les données et dans des langages de programmation comme C et C++, le programmeur doit gérer lui-même la manière dont il manipule ses données (notion de pointeurs). En Eiffel, tout est simplifié et tout est traité uniformément comme des objets.

L'objet se manipule via un « lien » (qu'on appelle en informatique une référence ou un pointeur). Une représentation dans la vie courante de ce « lien » est l'utilisation d'une télévision (l'objet) via la télécommande (le « lien », référence ou pointeur) . Quand vous utilisez la télécommande, vous êtes connecté (« lié ») à la télévision et pouvez changer de chaîne, modifier le son,... Ce que vous manipulez est la télécommande et non la télévision. Si vous bougez dans la pièce, vous vous déplacez avec la télécommande et pas avec la télévision tout en continuant à la manipuler[1].

Définissons une nouvelle classe appelée A. La classe A possède (i) un attribut *value* de type

1 Cette image provient de l'excellent livre « Thinking in Java » de Bruce Eckel et j'ai librement traduit les « handles » par des « liens »

INTEGER, (ii) un constructeur *with* qui permet d'initialiser *value,* (iii) une fonction *add* qui incrémente *value* de la valeur spécifiée dans l'argument et (iv) une fonction g*et_value* pour accéder au contenu de *value*.

```
class A

creation
    with

feature {ANY} --Constructor
    with(i : INTEGER) is
          do
                value := i
          end

feature {ANY}

    add(increment : INTEGER) is
          do
                value := value + increment
          end

    get_value : INTEGER is
          do
                Result := value
          end

feature {NONE} -- private
    value : INTEGER

end -- end of class B
```

La classe A est utilisée dans la classe TEST présentée ci-dessous.

```
1  class TEST
2
3
4  creation
5     make
6
7  feature
8
9  make is
10    local
11       i,j : A
12    do
13       create i.with(5)
14       j := i
15       i.add(4)
16
17       io.put_integer(i.get_value); io.put_new_line
19       io.put_integer(j.get_value); io.put_new_line
20    end
21
22 end -- end of class TEST
```

La première étape consiste à créer un objet i avec une valeur de 5 au moyen du mot clé **create**. Puis, on assigne l'objet i à l'objet j. Enfin, on applique le message *add(4)* à l'objet i. Le reste de la routine

make est identique et ne sert que pour l'affichage de value dans les objets i et j (appel du message *get_value* sur l'objet i). Pour compiler cet exemple, tapez

```
se c test.e -o test -clean
```

puis lancez-le en tapant

```
./test
```

Le résultat s'affiche:

```
9
9
```

De façon surprenante, le programme test affiche la même valeur 9 pour i.value et j.value alors qu'on attendrait i égal à 9 et j à 5. Pourquoi?

A la ligne 13, l'objet i est créé avec **create**, qui réserve de la place mémoire pour stocker les attributs de l'objet i. La ligne 14 « j := i » ne provoque pas la copie de l'objet i dans l'objet j, mais simplement, affecte à l'objet j, un «lien» vers l'objet i (une référence dans le jargon informatique). Ce « lien » permet à l'objet j de retrouver l' emplacement mémoire où se situe l'objet i.

Remarque: Si l'exemple de la télévision et de la télécommande vous semble trop éloigné de l'informatique. Cette opération de « lien » est équivalente à la création d'un « raccourci » ou « alias » d'un fichier dans un système multi-fenêtres (type KDE ou GNOME).

Par conséquent, lorsque l'objet i est modifié (ici par *add(4)),* l'objet j est toujours « lié » à i et donc à l'appel de la routine *j.value*, j « remonte » le lien, trouve l'objet i et affiche la valeur de l'attribut stocké dans i.

Vocabulaire: On dit que A est une classe de type *reference.* Une telle classe (mode par défaut en Eiffel et dans la plupart des langages à objets) nécessite sa création explicite au moyen du mot clé **create** en Eiffel.

6. Les objets *expanded*

Dans la section précédente, l'objet est créé puis manipulé via un « lien » (ou référence). L'avantage de cette technique est d'éviter de dupliquer plusieurs fois le même objet dans l'espace mémoire pour limiter la saturation de celle-ci. Toutefois, dans certains cas, on n'a aucun intérêt à utiliser un système de « liens ». Par exemple, pour des raisons d'optimisation, le système de « liens » pour les nombres (INTEGER, REAL, DOUBLE,...) n'est pas intéressant. C'est pourquoi, en Eiffel, il existe deux types de classes dites *expanded* et *reference*.

6.1. Déclaration et création

Au contraire de la classe de type *reference* qui nécessite une création via **create**, la classe de type *expanded* est prévue pour stocker directement l'objet.

```
un_reel : DOUBLE  -- Création d'un espace mémoire suffisant
                  --pour stocker un réel
un_reel := 16.0   -- Affectation de la valeur 16.0 à un_reel
un_reel.sqrt      -- Appel du message sqrt sur l'objet un_reel
```

Dans ce cas, on a créé un objet *un_reel* sans passer par le mot clé **create** et sans « lien ». Cette différence de stockage de l'objet dans l'espace mémoire entraîne parfois des erreurs.

6.2. Manipulation d'objets de type expanded

La classe TEST présente un exemple dans lequel on a deux variables INTEGER i et j qui sont des objets de classes de type *expanded*. On affecte le nombre entier 5 à i au moyen du signe « := ». Puis, on affecte l'objet i à j et enfin, on ajoute 4 à i. Les lignes faisant intervenir l'objet *io* (pour input/output) servent à l'affichage des résultats.

Compilez cet exemple « compile test.e -o test -clean », lancez-le avec « ./test » et commentez les résultats.

```
class TEST

creation
    make
feature
    i,j : INTEGER
    make is
    do
          i := 5
          j := i
          i := i + 4
          io.put_integer(i); io.put_new_line
          io.put_integer(j); io.put_new_line
    end
end — end of class TEST
```

7. Comparaison d'objets

7.1. Objets de type référence

Le programme TEST ci-dessous compare deux à deux trois objets atom1, atom2, atom3 de classe ATOM.

```
class TEST
creation
    make
feature {ANY}
    make is
          local
                atom1, atom2, atom3 : ATOM
          do
                create atom1.with(1.0,2.0,3.0)
                create atom3.with(1.0,2.0,3.0)
                atom2 := atom1
                if atom1 = atom2 then
                      io.put_string("atom1 = atom2%N")
                else
                      io.put_string("atom1, atom2 are different%N")
                end
                if atom1 = atom3 then
                      io.put_string("atom1 = atom3%N")
                else
```

```
                        io.put_string("atom1, atom3 are different%N")
                end
                if atom2 = atom3 then
                        io.put_string("atom2 = atom3%N")
                else
                        io.put_string("atom2, atom3 are different%N")
                end
        end
end -- end of class TEST
```

Après compilation et exécution, on s'aperçoit que seuls atom1 et atom2 sont égaux. Or, les objets atom1 et atom3 sont identiques, que se passe t-il? Le signe égal '=' sur des objets de type référence regarde simplement si les "liens" (adresses des objets) sont identiques et non pas si le contenu des objets est identique. En Eiffel, il faut redéfinir la routine is_equal dont la déclaration est la suivante:

```
is_equal(other : like Current) is
    do
    end
```

Pour la classe ATOM, deux atomes identiques signifie que leurs coordonnées sont égales, on peut implanter is_equal de la façon suivante:

```
is_equal(other : like Current) is
    do
        Result := (x = other.x) and (y = other.y) and (z = other.z)
    end
```

7.2. Objets de type expanded