

# Eiffel, SmartEiffel, conception par objets, programmation par objets, programmation par contrats, interface, généricité, liaison dynamique, simulation, et cætera.

Dominique Colnet – [colnet@loria.fr](mailto:colnet@loria.fr)

11 juillet 2007

**Avertissement** Ce document *en cours de rédaction* n'a pas la prétention d'être un ouvrage complet et exhaustif sur la conception et la programmation par objets avec le langage Eiffel. Ce n'est pas un manuel de référence du langage Eiffel ni la documentation du compilateur et des outils de SmartEiffel<sup>1</sup>. Il s'agit plutôt d'un ensemble de notes de cours, d'une suite plus ou moins pédagogique de transparents ainsi qu'une collection hétérogène d'exercices.

Comme c'est souvent le cas pour les photocopiés d'enseignement, le document actuel est constamment *en cours de rédaction*. La première version de ce document doit dater des années 1990<sup>2</sup>, date à laquelle nous avons décidé d'utiliser Eiffel pour la licence d'informatique de l'Université Henri Poincaré (Nancy 1) ainsi qu'à l'ESIAL (École Supérieure d'Informatique et Applications de Lorraine). Toutes les suggestions seront les bienvenues (même les corrections orthographiques).

Ce document peut être diffusé reproduit et/ou modifié librement à la seule condition d'indiquer sa provenance initiale et de laisser le présent avertissement inchangé. Si vous souhaitez disposer du texte source (L<sup>A</sup>T<sub>E</sub>X) du document, il suffit de me le demander par mail ([colnet@loria.fr](mailto:colnet@loria.fr)).

## Table des matières

<b>1 Les premiers pas</b>	<b>3</b>
1.1 Le premier programme . . . . .	4
<b>2 Introduction</b>	<b>10</b>
2.1 Bibliographie . . . . .	10
2.2 Historique et généralités . . . . .	12
<b>3 Classes et instances</b>	<b>16</b>
3.1 Classe (définition d'une catégorie d'objets) . . . . .	16
3.2 Instanciation (création d'objets) . . . . .	18
3.3 Types de base ( <b>expanded vs reference</b> ) . . . . .	23
3.4 Organisation mémoire . . . . .	24
3.5 Message et receveur . . . . .	25
3.6 Constructeurs . . . . .	28

---

<sup>1</sup>SmartEiffel The GNU Eiffel Compiler tools and libraries — <http://SmartEiffel.loria.fr>.

<sup>2</sup>Attention, n'en déduisez pas que le langage Eiffel n'a pas changé depuis 1990 ! En fait, le langage en est à sa troisième version et nous participons actuellement à la normalisation de cette nouvelle définition.

<b>4 Clients/Fournisseurs et interfaces</b>	<b>31</b>
4.1 Clients et fournisseurs . . . . .	31
4.2 Interface d'une classe . . . . .	33
<b>5 Comparer, manipuler et créer des objets</b>	<b>35</b>
5.1 Comparer deux objets . . . . .	35
5.2 Manipuler des triangles . . . . .	37
5.3 Ramasse-miettes . . . . .	40
5.4 Faire la copie d'un objet . . . . .	41
<b>6 Passage des arguments</b>	<b>46</b>
<b>7 ARRAY et STRING</b>	<b>47</b>
7.1 Classe STRING . . . . .	47
7.2 Interface de la classe STRING – short . . . . .	50
7.3 Classe ARRAY . . . . .	62
7.4 Interface de la classe ARRAY . . . . .	65
<b>8 Entrées sorties simples</b>	<b>71</b>
<b>9 Les assertions</b>	<b>73</b>
9.1 Invariant, pré- et post-condition . . . . .	73
9.2 Vérification ponctuelle . . . . .	74
9.3 Assertion pour les itérations . . . . .	74
9.4 Mise en œuvre . . . . .	74
9.5 Limites du mécanisme . . . . .	75
<b>10 Les outils</b>	<b>83</b>
10.1 Le mode emacs . . . . .	83
10.2 Les commandes du débogueur . . . . .	83
<b>11 Bien programmer en Eiffel</b>	<b>85</b>
<b>12 Routines à exécution unique</b>	<b>93</b>
<b>13 Divers</b>	<b>97</b>
13.1 Priorité des opérateurs . . . . .	97
13.2 Attributs constants . . . . .	99
13.3 Ancienne et Nouvelle notation pour l'instanciation . . . . .	101
<b>14 Exercices/anciens sujets d'examen</b>	<b>102</b>
<b>15 Généricité non contrainte</b>	<b>108</b>
<b>16 Héritage</b>	<b>112</b>
16.1 Spécialiser par enrichissement . . . . .	113
<b>17 Premières règles d'héritage</b>	<b>121</b>
17.1 Héritage des primitives . . . . .	122
17.2 Héritage des constructeurs . . . . .	122
17.3 Héritage de l'invariant . . . . .	122
<b>18 Contrôler l'héritage</b>	<b>122</b>
18.1 Renommer une méthode héritée . . . . .	123
18.2 Changer le statut d'exportation . . . . .	125

<b>19 Spécialiser par masquage</b>	<b>125</b>
19.1 Redéfinition d'une méthode héritée . . . . .	125
19.2 Héritage des assertions . . . . .	126
<b>20 Polymorphisme de variables et liaison dynamique</b>	<b>128</b>
20.1 Polymorphisme de variable . . . . .	128
20.2 Liaison dynamique . . . . .	132
<b>21 Relation d'héritage</b>	<b>134</b>
21.1 L'arbre d'héritage . . . . .	134
<b>22 Hériter pour implanter</b>	<b>139</b>
22.1 Cohérence locale – Cohérence globale . . . . .	142
<b>23 Classe abstraite</b>	<b>145</b>
<b>24 Généricité contrainte</b>	<b>147</b>
24.1 Limitation . . . . .	148
<b>25 Classifier en utilisant l'héritage</b>	<b>149</b>
<b>26 Héritage multiple</b>	<b>154</b>
<b>27 Comparer et dupliquer des objets</b>	<b>156</b>
27.1 Comparer avec <code>equal/is_equal</code> . . . . .	156
27.2 Dupliquer avec <code>clone/twin/copy</code> . . . . .	156
<b>28 Concevoir des programmes efficaces</b>	<b>160</b>
28.1 Utilisation de l'aliasing . . . . .	160
28.2 Éviter les fuites de mémoire . . . . .	163
<b>29 Appel de la super-méthode</b>	<b>165</b>
<b>30 Conclusion - Remarques</b>	<b>166</b>
<b>31 Diagrammes syntaxiques</b>	<b>170</b>

## 1 Les premiers pas

On suppose dans la suite que SmartEiffel est déjà correctement installé sur votre ordinateur. Une bonne façon de vérifier que l'installation est correcte consiste à taper dans une fenêtre de commande (fenêtre `command` DOS sous Windows ou dans une fenêtre `shell` sous Linux/Unix) la ligne suivante :

```
compile -o hi hello_world
```

En l'absence de message d'erreur, si vous tapez ensuite la commande :

```
hi
```

Le message `Hello World.` doit s'afficher sur l'écran. En cas de problème d'installation ou pour obtenir SmartEiffel, suivre le lien :

<http://SmartEiffel.loria.fr>

Ces pages permettent d'obtenir SmartEiffel, sa documentation en ligne, l'accès à la liste de diffusion électronique (*mailing list* `SmartEiffel@loria.fr`), le fichier FAQ<sup>3</sup> de SmartEiffel ainsi que beaucoup d'autres choses, comme par exemple l'interface de toutes les classes au format des navigateurs WEB (<http://SmartEiffel.loria.fr/libraries/classes.html>).

## 1.1 Le premier programme

Le premier programme présenté sur la vue 1 permet d'afficher le texte "Hello World." sur l'écran. En fait, ce programme très simple fait partie des programmes de démonstration fournis avec SmartEiffel et il est donc inutile de saisir ce programme dans un fichier, c'est déjà fait. Pour trouver le fichier de ce programme (i.e. le chemin d'accès complet sur le disque), utilisez comme suit la commande `finder` de SmartEiffel dans une fenêtre de commandes :

```
finder hello_world
```

De manière générale, la commande `finder` permet de trouver le chemin d'accès au fichier contenant la définition de n'importe quelle classe. Toutes les notations suivantes sont acceptées indifféremment :

```
finder INTEGER
finder integer
finder integer.e
```

Dans un programme Eiffel (la vue numéro 1 montre le programme correspondant à la classe `HELLO_WORLD`), les commentaires commencent par "--" et se terminent en fin de ligne. Les noms de classes sont en majuscules, par exemple `HELLO_WORLD` est un nom de classe. Il n'y a pas de syntaxe particulière pour le programme principal et pour faire une impression sur l'écran il faut déjà utiliser un des principes de la programmation par objets : application de l'opération `put_string` sur un objet référencé (pointé) par la variable `io`.

La typographie utilisée dans ce polycopié essaye de respecter les conventions suivantes : les mots réservés du langage Eiffel en **gras**, les commentaires en *italique* et les identificateurs sont écrits avec la police **ordinaire**.

---

<sup>3</sup>Foire Aux Questions - Frequently Asked Questions

```

class HELLO_WORLD
  -- My very first program in Eiffel.
  creation make
  feature
    make is
      do
        io.put_string("Hello World.%N")
      end
  end
end

```

Vue 1

Dans le texte de la chaîne de caractères "Hello World.%N", la séquence %N indique un passage à la ligne suivante. Pour les grands débutants, un bon exercice pour se familiariser avec Eiffel consiste à recopier le fichier `hello_world.e` chez soi, dans un répertoire de travail, puis, à modifier ce fichier pour essayer des variantes de ce programme. Attention : pour que votre propre copie de ce fichier soit prise en compte il faut lancer la commande de compilation dans votre répertoire de travail (le répertoire où se trouve votre fichier `hello_world.e`). N'hésitez pas à introduire volontairement des erreurs dans votre fichier pour vous familiariser avec les messages d'erreurs du compilateur. Pour savoir quels sont les fichiers Eiffel (\*.e) qui sont chargés par le compilateur, vous pouvez ajouter l'option `-verbose` lors du lancement de la compilation :

```

$ compile -verbose hello_world
SmartEiffel The GNU Eiffel Compiler
Release 1.1 (Charlemagne) (June 2003)
...
1      ...../SmartEiffel/tutorial/hello_world.e
2      ...../SmartEiffel/lib/kernel/any.e
3      ...../SmartEiffel/lib/kernel/platform.e
4      ...../SmartEiffel/lib/kernel/general.e
5      ...../SmartEiffel/lib/io/std_input_output.e
6      ...../SmartEiffel/lib/io/std_output.e
7      ...../SmartEiffel/lib/io/output_stream.e
8      ...../SmartEiffel/lib/io/std_input.e
9      ...../SmartEiffel/lib/io/input_stream.e
11     ...../SmartEiffel/lib/kernel/integer.e
12     ...../SmartEiffel/lib/kernel/integer_ref.e
13     ...../SmartEiffel/lib/kernel/comparable.e
14     ...../SmartEiffel/lib/kernel/numeric.e
15     ...../SmartEiffel/lib/kernel/hashable.e

```

```

16 ...../SmartEiffel/lib/kernel/string.e
17 ...../SmartEiffel/lib/kernel/boolean.e
18 ...../SmartEiffel/lib/kernel/boolean_ref.e
19 ...../SmartEiffel/lib/kernel/character.e
20 ...../SmartEiffel/lib/kernel/character_ref.e
21 ...../SmartEiffel/lib/kernel/native_array.e
22 ...../SmartEiffel/lib/basic/safe_equal.e
23 ...../SmartEiffel/lib/kernel/pointer.e
24 ...../SmartEiffel/lib/kernel/pointer_ref.e
...

```

Les différentes options que l'on peut passer au compilateur (commande `compile`) sont décrites en détail sur la page [http://SmartEiffel.loria.fr/man/compile\\_to\\_c.html](http://SmartEiffel.loria.fr/man/compile_to_c.html). Toutes les commandes (`compile`, `compile_to_c`, `finder`, `short`, `class_check`, `pretty`, `compile_to_jvm`, etc.) affichent également un petit résumé des options disponibles si on les lance avec l'option `-help`. Voici par exemple le résumé des options que l'on peut obtenir *en ligne* pour la commande `compile` :

```

$ compile -help
Usage: compile [options] <RootClass> <RootProcedure> ...
      or: compile [options] <ACEfileName>.ace
For information about and examples of ACE files, have a look
in the SmartEiffel/tutorial/ace directory.

```

Most of the following options are not available when using an ACE file.

Option summary:

Information:

```

-help           Display this help information
-version        Display SmartEiffel version information
-verbose        Display detailed information about what the
                compiler is doing

```

Warning levels:

```

-case_insensitive  Make class and feature names case-insensitive
-no_style_warning  Don't print warnings about style violations
-no_warning        Don't print any warnings (implies -no_style_warning)

```

Optimization and debugging levels (specify at most one; default is `-all_check`):

```

-boost           Enable all optimizations,
                but disable all run-time checks
-no_check        Enable Void target and system-level checking
-require_check   Enable precondition checking (implies -no_check)
-ensure_check    Enable postcondition checking (implies -require_check)
-invariant_check Enable class invariant checking (implies -ensure_check)
-loop_check      Enable loop variant and invariant checking
                (implies -invariant_check)
-all_check       Enable "check" blocks (implies -loop_check)
-debug_check     Enable "debug" blocks (implies -all_check)

```

C compilation and run-time system:

```

-cc <command>    Specify the C compiler to use
-cecil <file>    Take CECIL information from <file>
                (may be used more than once)
-o <file>        Put the executable program into <file>
-no_main         Don't include a main() in the generated executable

```

```

-no_gc           Disable garbage collection
-gc_info        Enable status messages from the garbage collector
-no_strip       Don't run "strip" on the generated executable
-no_split       Generate only one C file
-sedb           Enable the internal debugger
-wedit          Include support for the Wedit debugger
-clean          Run the "clean" command at the end

```

```

% compile -o hi hello_world
% hi
Hello World !

```

<http://SmartEiffel.loria.fr>

```

compile -o hi hello_world -verbose
compile -o hi hello_world -verbose -boost
Vue 2  compile -o hi hello_world -verbose -boost -O3
finder hello_world
short integer
class_check *.e
pretty hello_world
compile_to_jvm hello_world
java hello_world
...

```

Les fichiers qui contiennent les classes (les programmes) doivent être suffixés par ".e". Pour que le compilateur puisse facilement trouver les classes que vous utilisez, il faut également que l'on retrouve le nom de la classe dans le nom du fichier. Par exemple, le fichier contenant la classe "TOTO" doit obligatoirement s'appeler "toto.e". Sur les vues, ce qui est en **gras** est tapé par l'utilisateur. Pour compiler il faut impérativement donner en argument dans l'ordre, la classe racine (**toto**) ainsi que le nom de la procédure qui fait office de programme principal (la procédure **debut** sur l'exemple 3). Le langage C sert de langage cible et une compilation Eiffel (correcte) se termine par l'appel du compilateur C (**gcc** sous UNIX, cf vue 4). Il est également possible de produire du *byte-code* Java (cf. commande `compile_to_jvm`).

Vue 3

```

class TOTO -- Un autre exemple simple pour commencer.
creation debut
feature
  debut is
    local
      reponse: STRING
    do
      io.put_string("Oui ou non ? ")
      io.read_line -- Lecture d'une ligne sur le clavier.
      reponse := io.last_string
      inspect
        reponse
      when "oui", "o" then
        io.put_string("Oh oui !%N")
      when "non", "n" then
        io.put_string("Ben non.%N")
      else
        io.put_string("J'ai des doutes.%N")
      end -- inspect
    end -- debut
end -- TOTO

```

Vue 4

```

% compile -verbose TOTO debut
...
Type inference score : 100.00%
Done.
C compiling using "toto.make" command file.
System call "gcc -pipe -c toto10.c"
System call "gcc -pipe -c toto9.c"
System call "gcc -pipe -c toto8.c"
...
System call "gcc -pipe -c toto1.c"
System call "gcc -pipe toto1.o toto2.o ... toto10.o "
System call "strip a.out"
C code not removed.
Done.
% a.out

```



```

% ls -l a.out
-rwxr-xr-x 1 test01 invit 41248 Oct 18 08:53 a.out
% compile -verbose -o xx -boost -O3 TOTO debut
...
...
Type inference score : 100.00%
Done.
C compiling using "toto.make" command file.
System call "gcc -pipe -O3 -o xx toto.c"
System call "strip xx"
C code not removed.
Done.
% ls -l a.out
-rwxr-xr-x 1 test01 invit 16384 Oct 18 08:54 a.out

```

Vue 5

La commande de compilation `compile` admet des options comme par exemple l'option `-boost` qui provoque entre autres choses la suppression du code correspondant aux assertions. Cette option est vivement déconseillée en phase de mise au point car elle supprime aussi la trace d'exécution affichée habituellement en cas d'arrêt brutal. Certaines options comme `-O2` par exemple sont transmises au compilateur C. La description complète des autres options ainsi que toutes les autres informations qui concernent SmartEiffel sont accessibles via :

<http://www.loria.fr/SmartEiffel>

SmartEiffel est également disponible avec les distributions Linux sur CD rom (bien vérifier que le numéro de la version utilisée). L'option `-version` disponible sur toutes les commandes (`compile`, `finder`, `short`, `class_check`, `pretty`, etc.) permet de s'assurer du numéro de la version qu'on utilise.

Pour apprendre rapidement et facilement les principes essentiels d'Eiffel, nous vous conseillons de parcourir les exemples commentés qui sont dans la distribution de SmartEiffel (répertoire `SmartEiffel/tutorial`). Nous vous proposons de lire et d'essayer dans l'ordre les exemples suivants :

- `SmartEiffel/tutorial/hello_world.e` : le premier programme à tester ; n'hésitez pas à travailler sur une copie de ce programme dans un répertoire temporaire vous appartenant ;
- `SmartEiffel/tutorial/triangle/version*` : une progression à l'usage des grands débutants sur le thème des points et des triangles ; les quatre versions de la classe `TRIANGLE` sont accompagnées d'exercices permettant de présenter l'essentiels des bases ; une progression à ne pas manquer pour bien démarrer en Eiffel ;
- `SmartEiffel/tutorial/parking/` : petite simulation non graphique d'un parking pour voitures sur plusieurs niveaux ; il est également indispensable de regarder de près cet exercice car il présente bien le découpage en classes d'une application ;

- `SmartEiffel/tutorial/print_arguments.e` : pour savoir comment accéder aux arguments de la ligne de commande ;
- `SmartEiffel/tutorial/hanoi/` : simulation non graphique de l'exercice classique des tours de Hanoi ; à regarder car c'est un grand classique de la récursivité ;
- `SmartEiffel/tutorial/ace/` : pour savoir comment appeler le compilateur en choisissant classe par classe le niveau de vérification ; indispensable pour pouvoir se servir efficacement du débogueur `sedb-sedb` efficacement ;
- `SmartEiffel/tutorial/pyramid.e` : un exemple de récursivité avec retour en arrière (voir aussi `knight.e` et `pyramid2.e`) ;
- `SmartEiffel/tutorial/vision/*` : pour une visite des classes permettant d'écrire des application graphiques ;
- `SmartEiffel/tutorial/time/` : accès à l'heure et à la date ;
- `SmartEiffel/tutorial/basic_directory/` : manipulation des répertoires (voir aussi `directory/`) ;
- `SmartEiffel/tutorial/random/` : pour générer aléatoirement des nombres ;
- `SmartEiffel/tutorial/iterator/` : utilisation du patron de conception (*design pattern*) du même nom ;
- `SmartEiffel/tutorial/number/` : pour manipuler des grands nombres et des fractions ;
- `SmartEiffel/tutorial/external/C/` : pour savoir comment appeler des fonctions écrites en langage C à partir du langage Eiffel ;
- `SmartEiffel/tutorial/cecil/*` : pour savoir comment appeler des fonctions écrites en langage Eiffel à partir du langage C ;
- `SmartEiffel/tutorial/sorting/` : pour trier des COLLECTIONS ;
- `SmartEiffel/tutorial/memory/` : pour adapter le fonctionnement du ramasse-miettes au comportement de son application (réservé aux experts) ;
- `SmartEiffel/tutorial/tuple/` : pour manipuler facilement des n-uplets de valeurs ;
- `SmartEiffel/tutorial/agent/` : pour passer en paramètre du code à exécuter ultérieurement ;
- `SmartEiffel/tutorial/memory` : pour adapter le fonctionnement du ramasse-miettes au comportement de son application (réservé aux experts) ;
- `SmartEiffel/tutorial/scoop/` : pour distribuer son application sur plusieurs processeurs (plusieurs flots d'exécution – réservé aux experts) ;
- ...
- ...

## 2 Introduction

### 2.1 Bibliographie

Pour le débutant ne lisant pas l'anglais, nous conseillons vivement la lecture du livre "Cours de programmation par objets" cité sur la vue 6. Attention, certains autres ouvrages correspondent à la version 2.3 du langage Eiffel et non pas à la version 3.0 qui est celle que nous utilisons. Même s'il est en version 2.3, le livre de B.Meyer (vue 7) reste un excellent ouvrage d'initiation à Eiffel et est plus facile à lire que le manuel de référence du langage Eiffel version 3.0 (vue 6). D'autres références bibliographiques sont données dans le FAQ Eiffel (*Frequently Asked Questions* une copie de ce document est dans la distribution : `SmartEiffel/man/Eiffel.FAQ`).

Martine Gautier, Gerald Masini, Karol proch  
**Cours de programmation par objets**,  
Masson, 1995

Vue 6 Bertrand Meyer  
**Eiffel : The Language**, Prentice-Hall, 1991

Bertrand Meyer  
**Eiffel le Langage**, InterEdition, 1994

**Bibliographie : cf. Eiffel-FAQ**

### **Eiffel 2.3**

Bertrand Meyer  
**Object-Oriented Software Construction**,  
C.A.R. Hoare Series, Prentice-Hall International Series in  
Vue 7 Computer Science, Englewood Cliffs, New Jersey, 1988

Bertrand Meyer  
**Conception et programmation par objets, pour du logiciel  
de qualité**, InterEdition, Paris, 1990

**2.3 → 3.0 : Eiffel-FAQ L02**

## 2.2 Historique et généralités

Il faut dire *langage à objets* et ne plus utiliser la traduction littérale (et laide) *langage orienté objets* (vue 8).

Les non spécialistes du domaine ainsi que les magazines de vulgarisation (comme *Le Monde Informatique* par exemple) ne font aucune distinction entre *langage à objets* et *langage de classes*. Pour plus de détails sur ce sujet (historique des langages à objets et terminologie), consultez l'ouvrage cité sur la vue 11. Attention, ce livre ne contient que très peu de choses sur Eiffel (et en plus il s'agit de la version version 2.3 du langage Eiffel).

### Les langages à objets

(Object Oriented Languages)

=

langages de classes

∪

langages de frames

∪

langages d'acteurs

∪

langages hybrides

Vue 8

## **langages de classes**

*Domaine d'utilisation très général*

Vue 9 Simula (1973), Smalltalk (1972-76-80), C++ (1982-86), Objective-C (1984-86), Eiffel (1987-88), Flavors (1980), CLOS (1987), Object Pascal (1984-86), Pascal Turbo V5.5, Ada 9X (95-96), Java (95-96), ...

## **langages de frames**

*Réseaux sémantiques et travaux de Marvin Minsky ; utilisés en Intelligence Artificielle (IA)*

## **langages d'acteurs**

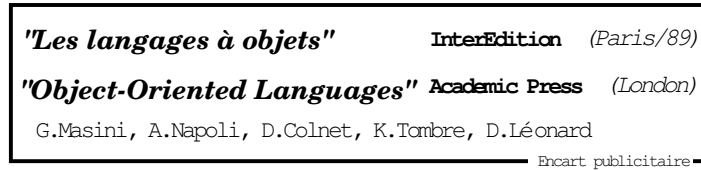
Vue 10 *Modèle pour la programmation parallèle*

## **langages hybrides**

*Mélange de caractéristiques provenant des trois familles précédentes et parfois même d'autres familles ...*

langages à objets  $\simeq$  langages de classes

Vue 11



Chapitre 2 : "Les classes" pp. 33-65

Section 6.3 : "Eiffel (2.3)" pp. 223-234

## Qu'est-ce qu'un langage de classes ?

- définition de classes
- instanciation
- héritage et liaison dynamique

Vue 12

## Eiffel c'est :

- **un langage de classes** (classe, instanciation, héritage et liaison dynamique)
- **typage statique** (variables typées)

- Vue 13
- invariant
  - pré-conditions
  - post-conditions
  - généricité
  - exceptions
  - routines à exécution unique
  - ...

*Bertrand MEYER (inventeur d'Eiffel*

Vue 14

```
class TOTO2
creation make
feature {ANY}
  make is
    local
      i: INTEGER
    do
      from
        i := 5
      until i = 0
      loop
        io.put_string("Bonjour Eiffel !%N")
        i := i - 1
      end
    end
end
end
```

Un compilateur est un programme qui n'est **jamais** terminé : il reste toujours des erreurs à corriger. Afin de nous faciliter la tâche je vous demande de suivre la procédure suivante le cas échéant :

1. Vérifiez que l'erreur est bien une erreur du compilateur et qu'il ne s'agit pas d'une erreur de votre part (relisez attentivement les messages émis par le compilateur avant de conclure).
2. Si vous travaillez ailleurs que sur les machines du réseau d'enseignement, vérifiez que le numéro de version est correct (version suffisamment récente).
3. Vérifiez que l'erreur en question n'est pas déjà répertoriée dans le couarail local rubrique Eiffel.
4. Isolez le bug, c'est-à-dire écrivez un programme reproduisant l'erreur, aussi petit que possible (le programme). Puis, postez une copie de ce programme dans l'outil de rapport d'anomalies (**SmartZilla**), disponible à partir des pages <http://SmartEiffel.loria.fr>.

## 3 Classes et instances

### 3.1 Classe (définition d'une catégorie d'objets)

Eiffel est complètement fondé sur la notion de classe et d'instance (vue 15). La classe s'apparente à un type abstrait de données et décrit un ensemble d'objets partageant la même structure et le même comportement. Elle sert de moule pour générer ses représentants physiques, les objets, également appelés instances. Une instance est toujours liée à une classe particulière, celle qui a servi de modèle lors de l'instanciation (la création de l'objet). En Eiffel, aucune nouvelle classe n'apparaît lors de l'exécution du programme. Contrairement à d'autres langages, en Eiffel, toutes les classes sont connues au moment de la compilation. La notion d'instance n'a de sens que lorsque le programme s'exécute (autrement dit, les objets ne sont créés qu'au moment de l'exécution). Programmer en Eiffel consiste à définir de nouvelles classes.

La notion de classe en Eiffel n'est pas sans rapport avec la notion plus mathématique et plus générale de type. Les attributs d'une classe (vue 16) correspondent à la partie structure d'un type implanté avec un produit cartésien. Les routines correspondent aux opérations du type. Elles manipulent les attributs et caractérisent les actions pouvant être effectuées par les objets (instances) de la classe. Dans le contexte d'un logiciel écrit en Eiffel, on peut parler indifféremment d'une classe ou d'un type (les deux mots sont synonymes).

La notion de classe prend en compte la notion de mémoire informatique et il faut décider, pour une classe donnée, ce qui sera mémorisé (attributs) et ce qui sera calculé (routines). Sur la vue 17, seules les entêtes des routines sont représentées. Ces routines utilisent les valeurs des attributs  $x$  et  $y$  de l'instance courante.



**CLASSE (ou TYPE)** : *description statique d'une famille d'objets ayant même structure et même comportement*

Vue 15 **INSTANCE (ou OBJET)** : *entité créée dynamiquement, en respectant les plans de construction donnés par sa classe*

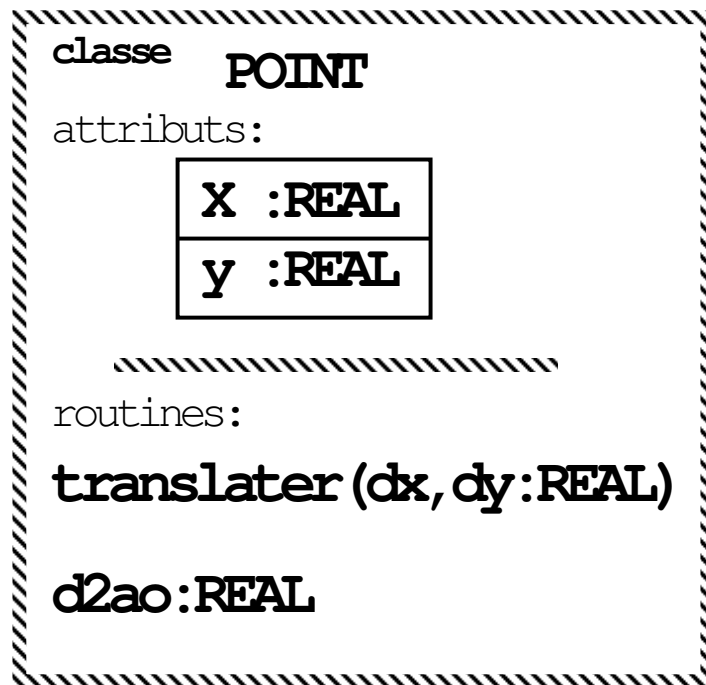
### Définir une classe

**Attribut(s)** *La partie données caractérisant à un instant particulier l'état d'une instance de la classe*

Vue 16

**Routine(s)** *Procédures ou fonctions spécialisées dans la manipulation des instances de la classe*

Vue 17



### 3.2 Instanciation (création d'objets)

L'instanciation consiste à créer une instance à partir du modèle donné par la classe. La classe sert de modèle pour construire les instances et seules les données (les attributs) sont dupliquées en mémoire.

La vue 19 montre le texte source Eiffel de la classe **POINT**. Les variables **x** et **y** sont les attributs de la classe **POINT**. Ce ne sont pas des variables globales<sup>1</sup> car chaque objet de la classe **POINT** disposera de ses propres valeurs pour les attributs **x** et **y** (pour cette raison, on dit également que **x** et **y** sont des *variables d'instances*). Ainsi, la procédure **translater** (toujours sur la vue 19) modifie l'instance courante (cette procédure modifie les variables propres à l'objet courant, les variables de l'instance courante). L'opérateur **^** utilisé dans la fonction **d2ao** est la notation Eiffel pour l'élévation à la puissance.

La procédure **essai\_point** (vues 20 et 21) montre comment il est possible d'utiliser la classe **POINT** de la page 19. Prise globalement, la procédure **essai\_point** ne fait pas grand chose d'intéressant (et là n'est pas la question). On va s'intéresser au déroulement au pas à pas des instructions de cette procédure afin de comprendre l'effet (la *sémantique*) d'instructions Eiffel élémentaires comme l'instanciation et l'affectation par exemple. Notons avant de commencer le déroulement étape par étape de cet exemple que la déclaration de variables locales à une routine est faite en utilisant le mot clé **local**. Comme son nom l'indique, une variable locale n'est accessible que localement, uniquement à l'intérieur de la routine correspondante.

Au point d'arrêt  $\odot$  1 (vue 22), les variables locales de la procédure **essai\_point** sont allouées dans la pile et initialisées automatiquement (oui, ce n'est pas comme en langage C

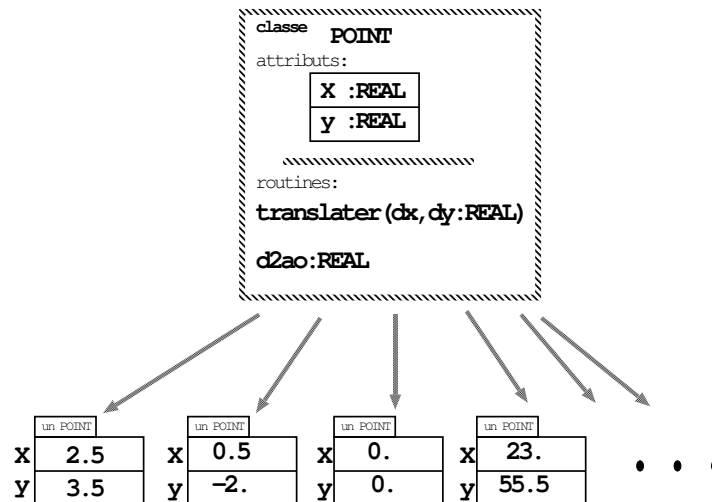
<sup>1</sup>Dans les anciens langages, C ou Pascal par exemple, une variable est dite *globale* si cette variable est unique et accessible globalement, c'est à dire depuis n'importe quel endroit du programme.

pour ceux qui connaissent). L'initialisation est différente pour une variable déclarée `REAL` et `POINT`. La notation `Void` en Eiffel est l'équivalent de `nil` en Pascal (ou `NULL` en C). La notation `Void` indique l'absence de valeur, c'est à dire le fait qu'une variable ne pointe sur aucun objet. La première instance de la classe `POINT` n'apparaît qu'en  $\odot$  2 (vue 22). Ainsi, la place allouée dans la pile pour une variable de type `POINT` est la place suffisante pour un pointeur et non pas pour une instance de la classe `POINT`. Au point  $\odot$  3 (vue 22), le point référencé par `p1` vient d'être modifié par l'intermédiaire de la procédure `translater`. Au point  $\odot$  6 (vue 23), une deuxième instance de la classe `POINT` fait son apparition en mémoire. L'affectation simple ne provoque jamais d'instanciation<sup>2</sup> et, au point  $\odot$  7 (vue 24), c'est une recopie de pointeur qui vient d'être effectuée et les variables `p2` et `p3` désignent (pointent) le **même** objet. Au point  $\odot$  8, la variable `p1` ne pointe plus aucun objet et l'instance allouée au point  $\odot$  2 est perdue pour tout le monde car inaccessible.

En fait, une instance n'est jamais perdue pour tout le monde car il existe un *ramasse-miettes*. Le travail du ramasse-miettes (*garbage collector*) consiste à récupérer la place mémoire occupée par les objets inaccessibles. La place mémoire récupérée est à nouveau disponible pour de nouvelles instanciations.

Pour une bonne compréhension d'Eiffel (et des langages à objets en général), il est indispensable de bien comprendre les explications données sous la forme de schémas mémoire (série de vues 20, 21, 22, 23 et 23). Un très bon exercice consiste à redessiner vous même sur une feuille de papier les différentes étapes de cette exécution. Si vous êtes à côté d'un ordinateur au moment de cet exercice, vous pouvez vous aider en utilisant le débogueur de SmartEiffel<sup>3</sup> qui permet de suivre l'exécution au pas par pas.

Vue 18



<sup>2</sup>Sauf si l'on considère les affectations d'un type `expanded` dans type référence compatible. Comme la manipulation des classes `expanded` est à mon avis une affaire de spécialistes, je préfère laisser ce cas particulier de côté pour l'instant.

<sup>3</sup>Pour exécuter un programme sous le débogueur de SmartEiffel, il suffit de le recompiler en ajoutant l'option `-sedb` puis de le lancer comme d'habitude.

Vue 19

```

class POINT
feature
-- Les attributs (variables d'instance):
  x, y : REAL
-- Les routines (méthodes):
  translater (dx, dy : REAL) is
    do
      x := x + dx
      y := y + dy
    end; -- translater
  d2ao : REAL is
    -- Distance au carré par rapport à l'origine !
    do
      Result := x^2 + y^2
    end; -- d2ao
end -- POINT

```

### Utilisation de la classe POINT

Vue 20

```

...
essai_point is
  local
    p1, p2, p3: POINT
    d: REAL
  do -- ⓪ 1
    -- Instanciation d'un point désigné par p1 :
    create p1 -- ⓪ 2
    -- Translation du point désigné par p1 :
    p1.translater(2,3) -- ⓪ 3
    -- Calcul dans d du nouveau d2ao de p1 :
    d := p1.d2ao -- ⓪ 4
  end

```

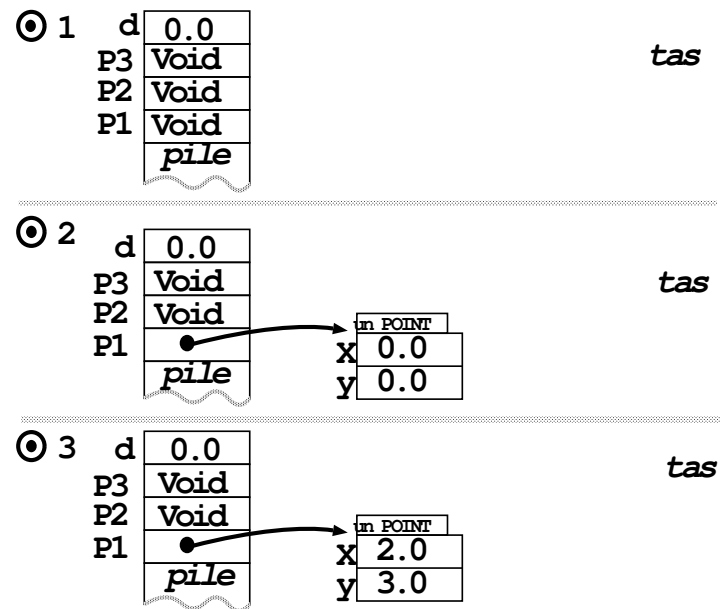
Vue 21

```

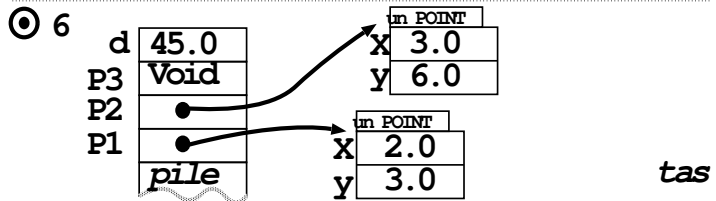
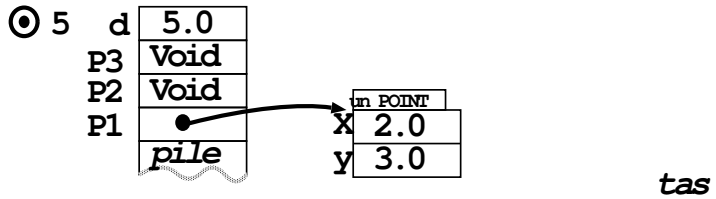
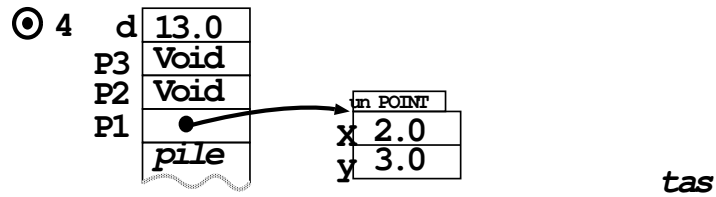
-- Consultation d'attributs :
d := p1.x + p1.y -- ⓐ 5
-- Instanciatiion et utilisation :
create p2
p2.translater(p1.y,6)
d := p2.d2ao -- ⓑ 6
-- Signification de l'affectation :
p3 := p2
p3.translater(1,6) -- ⓒ 7
-- Perdre l'accès à une instance :
p1 := Void -- ⓓ 8
-- Affectation d'une variable contenant Void :
p2 := p1 -- ⓔ 9
end -- essai_point
...

```

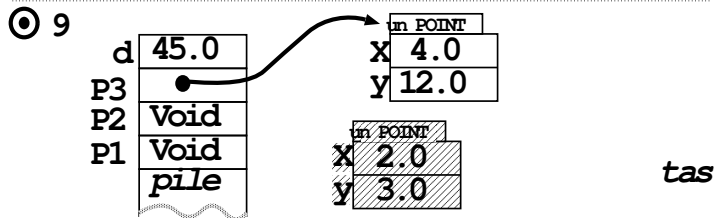
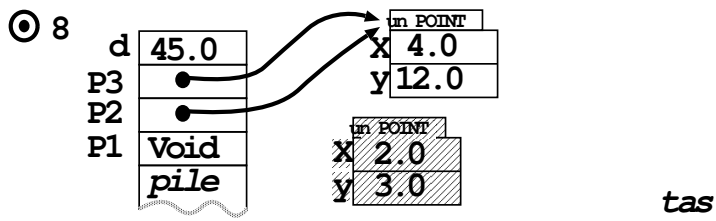
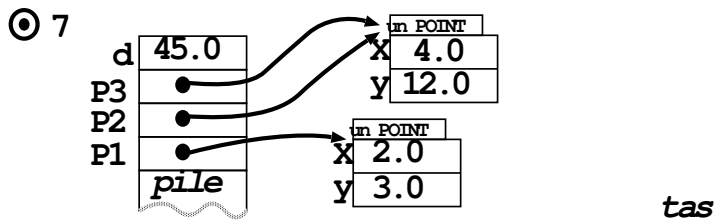
Vue 22



Vue 23



Vue 24





Vue 26

Même règle d'initialisation pour les variables locales et pour les attributs.	<code>unTableau : ARRAY [INTEGER]</code>	Void
	<code>uneListe : LIST [REAL]</code>	Void
	<code>un_fichier : STD_FILE_READ</code>	Void
	<code>unePhrase : STRING</code>	Void
	<code>unPoint : POINT</code>	Void
	<code>    x : REAL</code>	0.0
	<code>    i : INTEGER</code>	0
	<code>    c : CHARACTER</code>	'null'
<code>    b : BOOLEAN</code>	false	

### 3.4 Organisation mémoire

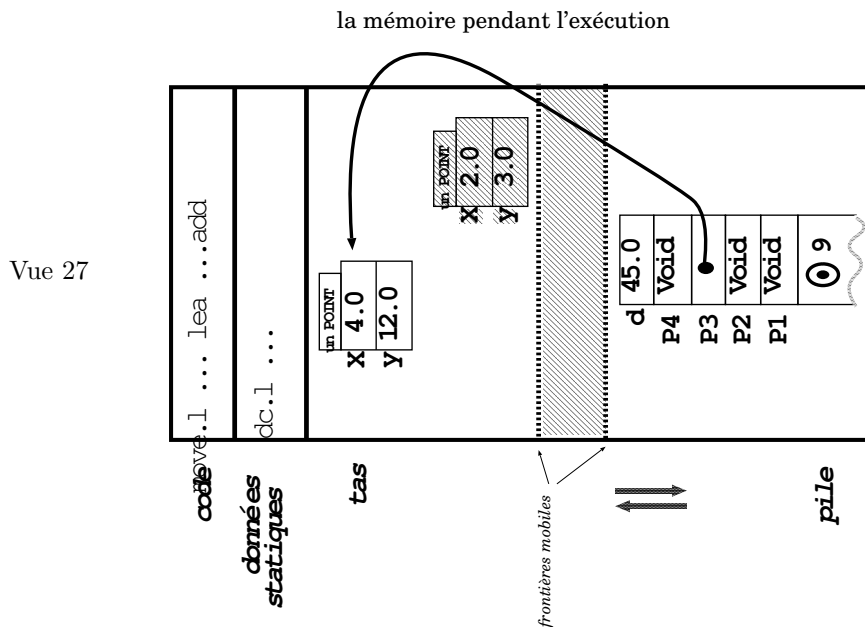
A l'exécution, de façon simpliste, on peut considérer que l'organisation en mémoire d'un programme Eiffel correspond schématiquement à l'organisation présentée sur la vue 27. La zone de mémoire associée au programme à l'exécution est divisée en quatre zones.

- La zone *code* contient le code machine des routines (procédures et fonctions) des différentes classes du programme.
- La zone *données statiques* (ou *data* en Anglais) contient tous les objets dont la place mémoire doit être réservée avant l'exécution (variables de la procédure servant de programme principal et routines **once**).
- La zone *tas* (ou *heap* en Anglais) contient toutes les instances devant être allouées durant l'exécution. Cette zone comporte des parties libres et des parties occupées au gré des instanciations et des restitutions d'instances. C'est dans cette zone que le ramasse-miettes fait son travail de récupération mémoire.
- La zone *pile* (ou *stack*) qui permet l'interprétation du programme en permettant de mémoriser les points de retour pour l'exécution des routines (la récursivité est toujours possible). C'est aussi dans cette pile que sont allouées les variables locales des routines.

Si tout se passe bien, la pile ne déborde jamais dans le tas et le tas n'est jamais plein. Notons que cette vision schématique de la mémoire est également valable pour la très grande majorité des langages informatiques : Pascal, Lisp, Smalltalk, Ada, C, C++, C#, Caml, VisualBasic, Java, etc. (tous les langages applicatifs en général). Même si cela n'est pas essentiel pour un débutant, un développeur confirmé doit garder à l'esprit cette vision de l'organisation mémoire afin de bien maîtriser son logiciel, et en particulier, pour savoir où sont stockées ses informations.



**Exercice 1** En Pascal ou en C, donnez un exemple d'allocation dans le tas, un exemple d'allocation dans la pile et un exemple d'allocation dans la zone statique.



### 3.5 Message et receveur

Dans le jargon de la programmation par objets, l'instruction Eiffel de la vue 28 est un *envoi de message* à l'objet désigné par *p*. Dans cet exemple, le receveur est l'instance désignée par "p" (ou référencée par "p"), le message est "translater(2,3)". Enfin, on dit également que "translater(dx,dy : REAL)" est le sélecteur de cet envoi de message.

A l'exécution, c'est le type du receveur qui sert à choisir la routine à déclencher comme nous le verrons plus tard dans la section consacrée à la liaison dynamique (section 20 page 128).

Partant du principe qu'une routine sert à manipuler une seule catégorie d'objets bien précise (les objets de la classe où est définie la routine en question), il est impossible de déclencher l'exécution d'une routine sans préciser un receveur (i.e. sans donner l'objet *cible* sur lequel on souhaite appliquer l'opération). L'exécution du fragment de programme Eiffel de la vue 29 provoque donc une erreur lors de l'exécution. Comme il est souvent beaucoup plus difficile que sur l'exemple de la vue 29 de détecter statiquement (c'est-à-dire avant l'exécution) ce genre d'erreur, la plupart des compilateurs ne cherchent même pas à les détecter ! Aucun message d'erreur n'apparaît lors de la compilation du texte source de la vue 29. L'erreur sera détectée uniquement lors de l'exécution. Le message d'erreur est :

\*\*\* Error at Run Time \*\*\*: Call with a Void target.

Cette erreur est probablement l'erreur la plus courante pouvant survenir lors de l'exécution d'un programme Eiffel. Un bon exercice pour les débutants consiste à taper la classe de la vue 29 afin de reproduire le message d'erreur en question. A condition de ne pas avoir compilé

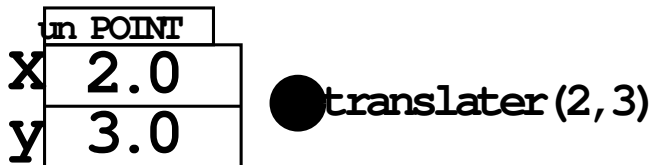
le programme en mode `-boost`, d'autres informations accompagnent ce message d'erreur et permettent de localiser très précisément la cause de l'erreur. Il faut rapidement s'habituer à ce message d'erreur ainsi qu'à l'utilisation du débogueur (en particulier la visualisation de la pile dans le contexte précis de l'erreur).

La vue 30 présente le vocabulaire habituellement utilisé pour Eiffel : primitive, routine, procédure, fonction et attribut. Il n'est pas inutile de connaître également les termes *méthode* et *variable d'instance* qui sont utilisés dans d'autres langages à objets comme Smalltalk ou Java par exemple. A mon avis, le vocabulaire de Smalltalk (variable d'instance et méthode) correspond mieux à l'esprit des langages de classes et surtout évite la confusion possible entre la notion de procédure Pascal et celle de procédure Eiffel<sup>1</sup>.

### message, receveur et sélecteur

`p.translater(2,3)`

Vue 28



<sup>1</sup>Une procédure Eiffel n'est pas dissociable de la classe où elle est définie alors qu'une procédure Pascal (ou une fonction C) est définie de façon globale, en dehors de tout contexte particulier.

## Pas de message sans receveur !

```

class LOUCHE
creation
debut
feature {ANY}
debut is
  local
    p : POINT
  do
    p.translater(2,3)
  end -- debut
end -- class louche

*** Error at Run Time ***: Call with a Void target.

```

Vue 29

## Vocabulaire Eiffel

**routine** : procédure ou fonction (ou *méthode*)  
**fonction** : routine avec un type pour le résultat  
**procédure** : routine sans type pour le résultat  
**attribut** : mémoire d'un objet (ou *variable d'instance*)  
**primitive** : routine ou attribut

Vue 30

### 3.6 Constructeurs

La clause **creation**, déjà utilisée pour indiquer la procédure servant de programme principal indique la liste des procédures pouvant servir de constructeurs pour une classe (vue 31).

Un constructeur (une procédure dont le nom figure derrière le mot clé **creation**) peut être utilisé en combinaison avec la notation “**create** ” qui marque l’instanciation (vue 32, ⊙ 2 et ⊙ 5) ou peut aussi être utilisé comme une procédure ordinaire ⊙ 4. Dans ce dernier cas, il n’y a pas d’instanciation. Dès qu’une classe comporte un (ou des) constructeur(s), il est interdit d’utiliser la notation d’instanciation seule (comme “**create p**” par exemple). Il faut impérativement passer par un constructeur.

**Exercice 2** Que dire du fragment de programme suivant lors de la compilation ? lors de l’exécution ?

```
...
p := Void
p.translater(2,3)
```

Le mot réservé **Current** permet de faire référence à l’objet receveur d’un message donné (on parle aussi de *cible*, vue 34). En accord avec le principe de la vue 29, **Current** n’est jamais **Void**.

**Exercice 3** Écrivez sans utiliser **Current** une deuxième version de la procédure **translater-DeuxFois** de la vue 34. Comparez les deux solutions.

**Exercice 4** Quel est votre avis sur le fragment de code suivant ? erreur à la compilation ? erreur à l’exécution ? défaillance (temporaire) de l’auteur du programme ?

```
...
if Current = Void then
  io.put_string("Bonjour")
else
  io.put_string("Oui")
end
...
```

## création/instanciation

Vue 31

```

class POINT
creation {ANY}
  make
feature {ANY} -- Les attributs :
  x, y : REAL
feature {ANY} -- Les routines :
  make(vx, vy : REAL) is
    do
      x := vx
      y := vy
    end -- make
  translater (dx, dy : REAL) is
    ...

```

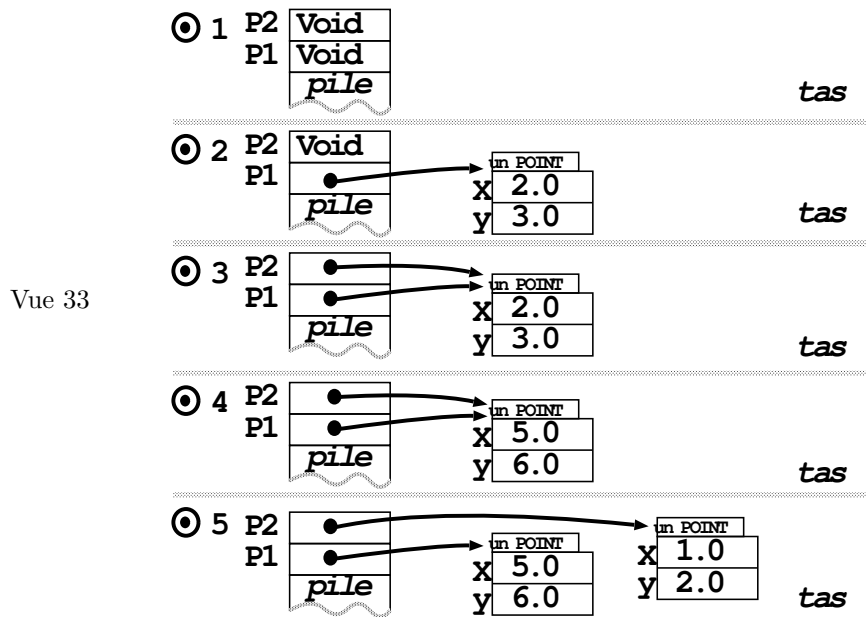
## Utilisation des constructeurs

Vue 32

```

...
local
  p1, p2 : POINT
do -- ⊙ 1
  -- Instanciation d'un point désigné par p1 :
  create p1.make(2,3) -- ⊙ 2
  -- Translation du point désigné par p1 :
  p2 := p1 -- ⊙ 3
  p2.make(5,6) -- ⊙ 4
  create p2.make(1,2) -- ⊙ 5

```



### Current pour désigner le receveur

Vue 34

```

class POINT
...
  translaterDeuxFois (dx, dy : REAL) is
    do
      Current.translater(dx,dy)
      Current.translater(dx,dy)
    end -- translaterDeuxFois
...
end -- class point

```

## 4 Clients/Fournisseurs et interfaces

### 4.1 Clients et fournisseurs

On peut définir un triangle à partir de trois points (vue 35). Une instance de la classe TRIANGLE possède trois attributs initialisés par défaut à `Void`. Notez que la procédure `translater` de la classe TRIANGLE utilise la procédure de même nom de la classe POINT (ce **n'est pas** récursif).

La définition de la classe TRIANGLE utilise le type POINT (vue 35). On dit alors que la classe TRIANGLE *est un client de* la classe POINT (vue 37). Réciproquement, on dit que POINT *est un fournisseur de* TRIANGLE.

```

class TRIANGLE
  creation {ANY}
    make
  feature {ANY}
    p1, p2, p3 : POINT
    make (vp1, vp2, vp3 : POINT) is
      do
        p1 := vp1; p2 := vp2; p3 := vp3
      end -- make
    translater (dx, dy : REAL) is
      do
        p1.translater(dx,dy)
        p2.translater(dx,dy)
        p3.translater(dx,dy)
      end -- translater

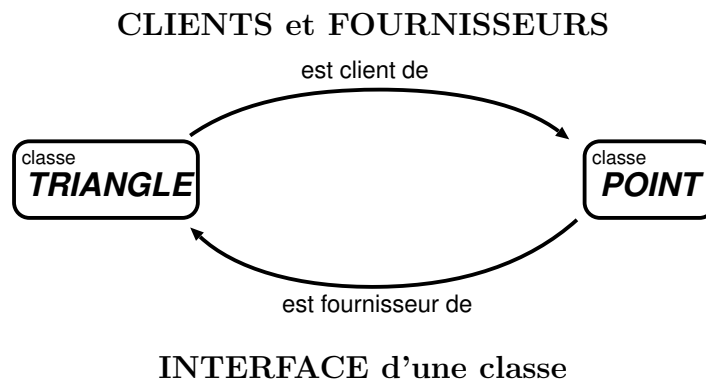
```

Vue 35

Vue 36

```
feature {NONE}
  melanger is
    -- Etrange procedure ...
    local
      tmp : POINT
    do
      tmp := p1
      p1 := p2
      p2 := tmp
    end -- melanger
end -- triangle
```

Vue 37





## 4.2 Interface d'une classe

L'*interface* d'une classe (ou partie publique d'une classe) correspond à l'ensemble des primitives utilisables par un client. La mention {ANY} derrière le mot clé **feature** indique que les primitives définies dans la suite sont utilisables par n'importe quel client. Ainsi, un client de la classe TRIANGLE (vue 35) peut utiliser toutes les primitives **p1**, **p2**, **p3**, **make** et **translate**. Les attributs (**p1**, **p2** et **p3**) ne sont pas exportés en écriture. Seule la consultation de ces attributs est possible. Un client ne peut jamais changer directement la valeur d'un attribut, il peut seulement lire la valeur (vue 39). Syntactiquement, pour un client, la consultation d'un attribut et l'appel d'une fonction sans argument sont identiques : on parle dans ce cas du *principe de référence uniforme*. Ce choix syntaxique est important car il permet de modifier l'implantation d'une classe sans rien changer du point de vue des clients (c'est à dire sans changer son interface). Ainsi, une valeur anciennement mémorisée dans un attribut peut devenir le résultat d'un calcul issu de l'exécution d'une fonction sans argument, et inversement, ceci sans modifier les programmes clients.

Inversement, la mention {NONE} indique que les primitives ne sont utilisables qu'à l'intérieur de la classe elle-même (partie privée de la classe). Dans une même classe, on peut mettre plusieurs clauses **feature** et ainsi exporter ou non une primitive. L'ordre dans lequel on donne les primitives est sans importance. La vue 36 complète le texte source de la classe TRIANGLE commencé à la vue précédente. Le compilateur vérifie qu'un client n'utilise que des primitives exportées (vue 38) et qu'il n'affecte pas d'attributs. Syntactiquement, l'écriture d'un attribut est impossible depuis une autre classe. En fait, le caractère '.' n'est jamais autorisé à gauche de l'opérateur d'affectation := (voir le diagramme *instruction* de la page ??). La seule façon d'autoriser l'écriture d'un attribut depuis une autre classe consiste à ajouter explicitement une procédure d'écriture et à exporter cette procédure<sup>1</sup>.

Finalement, toujours sur la vue 38, seul un receveur, instance de TRIANGLE peut utiliser **melanger**.

Entre les deux extrêmes que sont ANY et NONE il est possible de moduler l'exportation d'une primitive vers n'importe quelle classe. Par exemple, si on souhaite autoriser l'opération **melanger** dans le corps des routines de la classe POINT, on mentionne cette classe dans la clause **feature** correspondante (vue 40). Cet aspect est très important pour contrôler les interactions (on parle aussi de couplage) entre différentes classes (ou modules) d'un logiciel complexe<sup>2</sup>.

**Exercice 5** En Eiffel, même si la classe POINT possède un attribut **x**, l'instruction suivante est illicite : `point.x := 3.5`

1. Comment faut-il procéder pour arriver à ses fins sans que le compilateur Eiffel rejette le programme ?
2. Selon vous, pour quelle(s) raison(s) est-il interdit de modifier les attributs d'un objet depuis l'extérieur ?

<sup>1</sup>Tout comme C++, Java n'oblige pas le passage par une procédure d'écriture ce qui est une erreur à mon sens.

<sup>2</sup>L'absence d'un mécanisme sérieux d'exportation est un des graves défauts de Java.

## exportation des primitives

local  
 t : TRIANGLE  
 do  
 ...  
***t.melanger*** ← erreur à la compilation

Vue 38

```
class TRIANGLE
  ...
  Current.melanger
  ...
end -- triangle
```

...  
 local  
 pa : POINT  
 r : REAL  
 t : TRIANGLE  
 do  
 !!pa.creer(1,2)  
 r := pa.x  
***pa.x := 3***  
***t.p1 := pa***

Vue 39

ERREUR SYNTAXIQUE

```

class TRIANGLE
...
feature {POINT}
  melanger is
...
end -- triangle

```

## exporter sélectivement

Vue 40

```

class POINT
...
      t : TRIANGLE
      ...
      t.melanger
...
end -- triangle

```

## 5 Comparer, manipuler et créer des objets

### 5.1 Comparer deux objets

La notion de mémoire informatique impose de pouvoir distinguer le fait que deux objets soient identiques ou bien qu'il s'agisse du même et unique objet. Autrement dit, par exemple, si les objets que l'on manipule sont des voitures, il est possible de se demander si la voiture `v1` et la voiture `v2` sont deux voitures identiques (même marque, même couleur) ou bien s'il s'agit d'une seule et même voiture (la mienne par exemple).

Le problème de la comparaison d'objets est un problème finalement assez complexe que nous reprendrons en détail plus tard (en section 27 page 156). Pour l'instant et pour simplifier, toujours sur l'exemple des voitures, la fonction `is_equal` permet de savoir si les deux voitures sont de la même marque et de la même couleur alors que l'opérateur de comparaison `=` permet de savoir s'il s'agit d'une seule et unique voiture. Les vues 41 et 42 montrent l'utilisation de ces deux façon de comparer des objets en utilisant la classe `POINT`.

## Comparaison : is\_equal et =

```

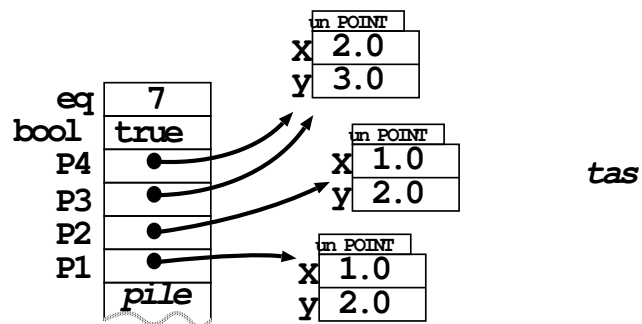
...
local
  p1, p2, p3, p4: POINT; bool: BOOLEAN; eq: INTEGER
do
  create p1.make(1,2)
  create p2.make(1,2)
  create p3.make(2,3)
  p4 := p3
  if p1.is_equal(p2) then
    bool := not p1.is_equal(p3)
  end
  if p1 = p2 then
    eq := 3
  elseif p3 = p4 then
    eq := 7
  end
end
-- ☉ cf. schéma mémoire sur la vue suivante

```

Vue 41

☉

Vue 42



## 5.2 Manipuler des triangles

Les vues 44, 45, 46 et 47 correspondent aux schémas mémoire pour les points d'arrêt du code Eiffel de la vue 43. Encore une fois, pour une bonne compréhension des mécanismes élémentaires des langages à objets, il est indispensable de bien comprendre la signification de ces schémas mémoire. Un exercice indispensable consiste à redessiner vous même sur une feuille de papier les différentes étapes de cette exécution. Si vous êtes à côté d'un ordinateur au moment de cet exercice, vous pouvez vous aider en utilisant le débogueur de SmartEiffel (`-sedb`) qui permet de suivre l'exécution au pas par pas.

A l'instant correspondant à la vue 46, une même instance de la classe POINT (le POINT [2.0,2.0]) est référencée par les deux triangles. L'instance [5.0,5.0] est référencée deux fois par le même triangle (on parle dans ce cas d'*aliasing*).

A la vue 47, deux instances, une de la classe POINT et une de la classe TRIANGLE ne sont plus accessibles du tout (ces instances sont en grisé sur cette vue). La place mémoire occupée par ces deux instances peut être alors récupérée par le ramasse-miettes dès que celui-ci se mettra en fonction. En général, le ramassage n'est pas immédiat car le déclenchement du ramasse-miettes dépend de l'utilisation mémoire totale.

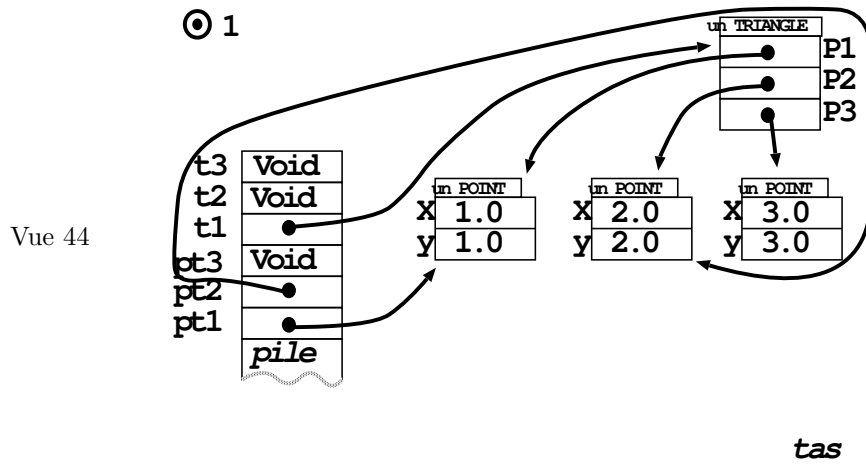
**Exercice 6** En utilisant uniquement le texte source de la vue 43, redessinez les schémas des vues 44, 45, 46 et 47.

**Exercice 7** Pendant l'exécution du fragment de programme de la vue 43, quel est le nombre total d'instanciations effectuées ?

```

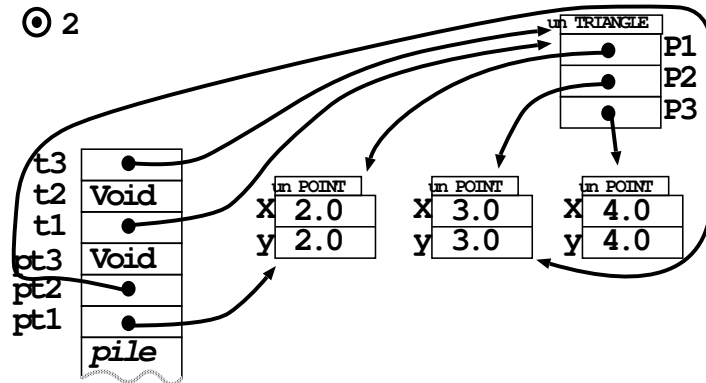
local
  pt1, pt2, pt3 : POINT
  t1, t2, t3 : TRIANGLE
do
  create pt1.make(1,1)
  create pt2.make(2,2)
  create pt3.make(3,3)
  create t1.make(pt1, pt2, pt3)
Vue 43  pt3 := Void -- ⊙ 1
        t3 := t1
        t3.translater(1,1) -- ⊙ 2
  create pt2.make(5, 5)
  create t2.make(pt2, pt1, pt2) -- ⊙ 3
  t2.translater(1, 1)
  pt2 := t1.p3
  t3 := Void
  t1 := t3 -- ⊙ 4

```



⊙ 2

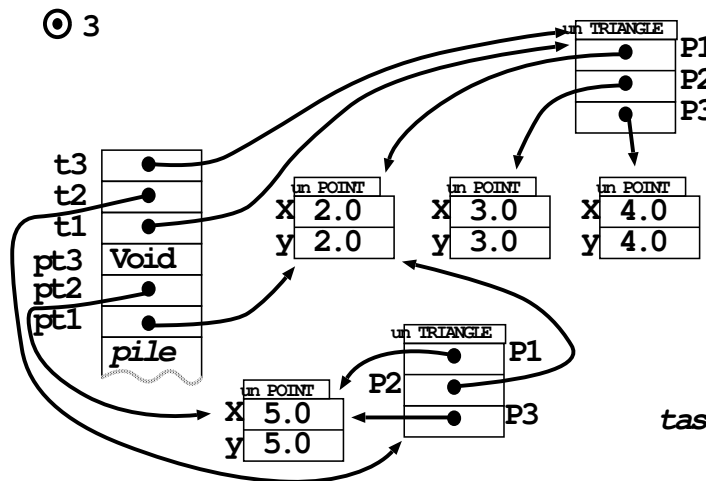
Vue 45



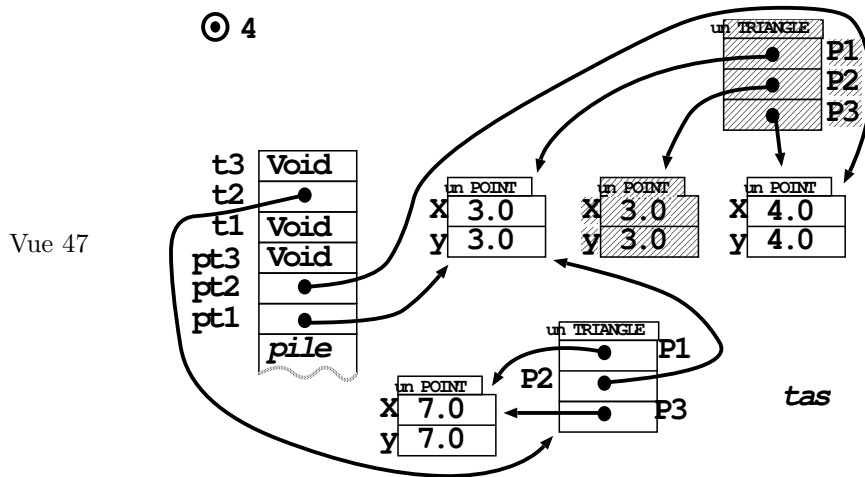
*tas*

⊙ 3

Vue 46



*tas*



### 5.3 Ramasse-miettes

Volontairement, Eiffel n'offre pas la possibilité au programmeur de restituer explicitement la mémoire occupée par une instance dans le tas (vue 48). Ce choix évite d'aboutir à la situation incohérente dans laquelle une zone mémoire du tas est supposée libre par le gestionnaire de mémoire alors qu'elle est encore référencée par l'utilisateur (cas d'un pointeur invalide, en anglais, *dangling pointer*).

Ramasse-miettes est le nom donné au programme chargé de collecter les zones mémoires libres dans le tas (les instances inaccessibles par exemple). Plusieurs variantes de ramasse-miettes existent :

- **incrémental** - s'il s'agit d'un ramasse-miettes qui fonctionne au fur et à mesure de l'exécution du programme utilisateur ; un tel ramasse-miettes n'attend pas que le tas soit complètement plein pour chercher des places libres et ne provoque pas d'interruption du programme utilisateur ; tout se passe comme s'il fonctionnait en parallèle.
- **déclenchable** - si le ramasse-miettes peut être déclenché explicitement par l'utilisateur ; en général il se déclenche lorsqu'une allocation survient et que la zone libre du tas est pleine.
- **débrayable** - s'il est possible de mettre hors service le ramasse-miettes pour éviter un risque d'interruption ; en Eiffel, le ramasse-miettes est débrayable.

Le ramasse-miettes de SmartEiffel n'est pas incrémental. Il peut être déclenché explicitement ou être débrayé (voir pour cela la classe MEMORY). En outre, pour les programmes sans *fuite de mémoire*, il est possible de ne pas intégrer de ramasse-miettes du tout (voir l'option `-no_gc` de la commande `compile`).



**Exercice 8** Quel genre de programme Eiffel peut-on écrire pour vérifier le bon fonctionnement du ramasse-miettes ? Écrire et faire fonctionner un tel programme pour savoir si le ramasse-miettes est effectivement en service.

Note 1 : pensez à utiliser l'option `-gc_info` pour vérifier que le ramasse-miettes est bien déclenché.

Note 2 : soyez prudent de ne pas écrouler le fonctionnement de la machine pendant votre essai !

Ramasse-miettes ou pas, ce n'est pas une raison pour gaspiller de la mémoire. Vous devez savoir exactement s'il y a lieu d'instancier (d'allouer) ou non !

### allocation/restitution

	Pascal	C	Eiffel
allocation	new	malloc	<b>create</b>
restitution	dispose	free	

Vue 48

### ramasse-miettes Eiffel

*Incrémental ?* oui/non

*Déclenchement ?* oui/non

*Débrayable ?* oui

## 5.4 Faire la copie d'un objet

Afin de laisser la possibilité de gérer au mieux la mémoire, il existe deux fonctions pour faire la copie d'objets (vue 49). La procédure `copy` sert à ré-initialiser un objet préalablement existant à partir d'un modèle donné. La fonction `twin` sert à créer un objet qui sera instancié pour l'occasion. En fait, comme nous le verrons en détail plus tard (page 156), la fonction `twin` appelle la procédure `copy` juste après l'allocation de la nouvelle instance.

**Exercice 9** Simuler l'appel de la fonction `twin` de la vue 53 en utilisant un constructeur de la classe `TRIANGLE` ainsi que la procédure `copy`.

**Exercice 10** Que se passe-t-il si on exécute l'instruction de la vue 53 en partant de l'état mémoire de la vue 52 ? Faire le schéma mémoire.

**Exercice 11** Que se passe-t-il si on exécute l'instruction de la vue 52 en partant de l'état mémoire de la vue 53 ?

**Exercice 12** Quel est le résultat si on applique la fonction `deep_twin` sur l'état mémoire de la vue 53 ? Faire le schéma mémoire.

**copy : la procédure**

```
t1.copy(t2)
```

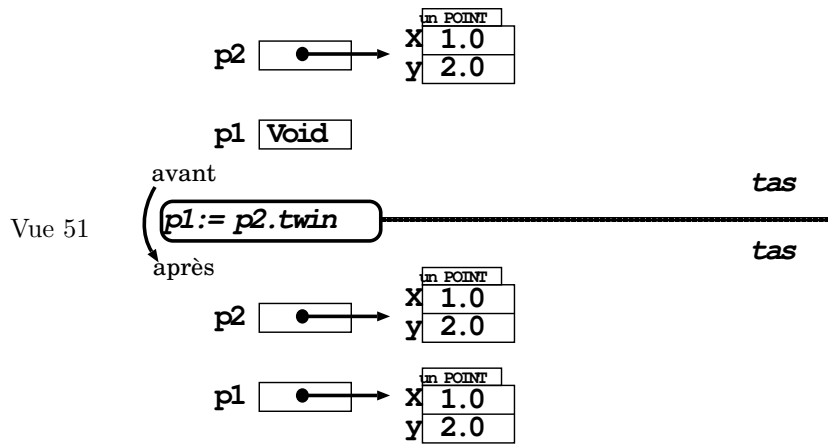
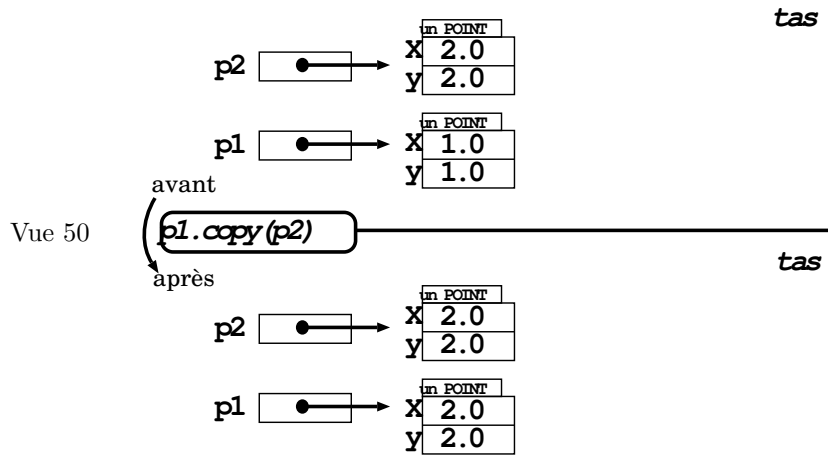
*Copier sans instancier (Modifie la cible)*

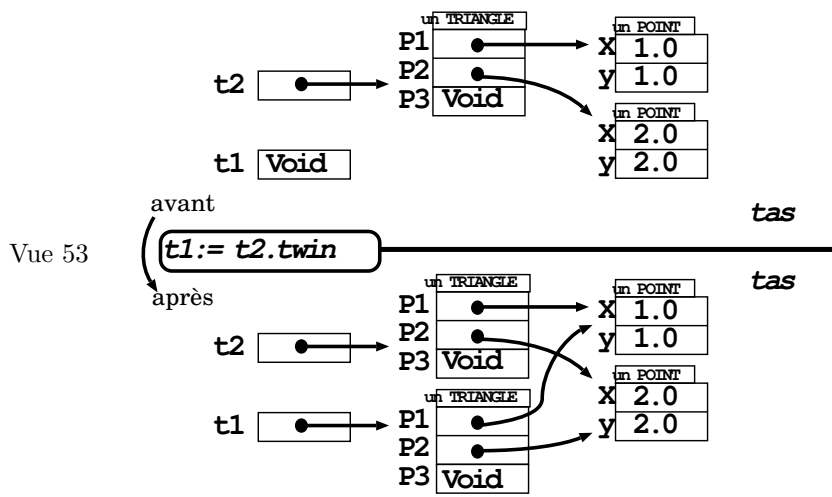
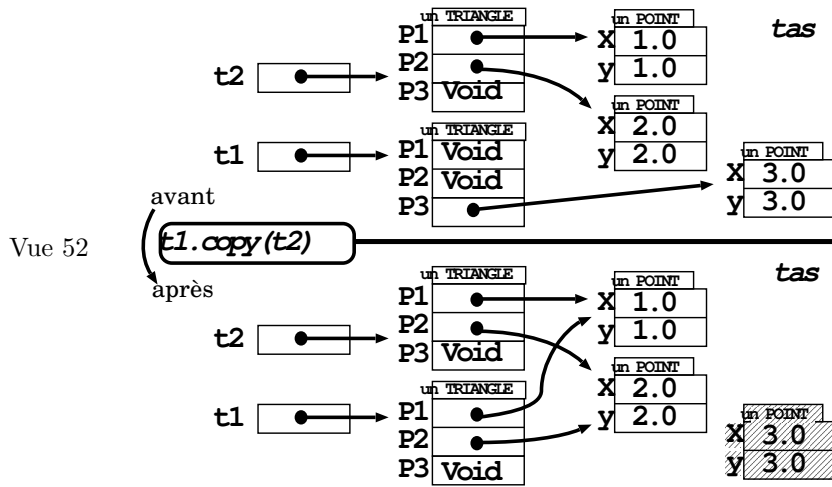
Vue 49

-----  
**twin : la fonction**

```
t1 := t2.twin
```

*Instancier puis copier (Retourne une nouvelle instance)*





## Copier en profondeur

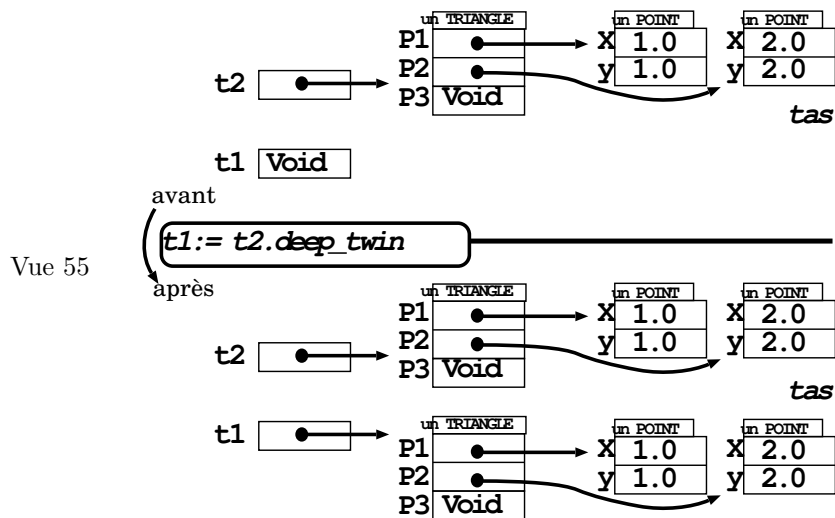
`deep_twin` : la fonction

```
t1 := t2.deep_twin
```

Vue 54

*Fabrique une copie n'ayant aucune zone mémoire commune avec le modèle initial*

**Attention au gaspillage !**



## 6 Passage des arguments

Les *arguments formels* sont les arguments utilisés dans la définition d'une routine (pA et y sur la vue 56). Les *arguments effectifs* sont ceux que l'on utilise pour appeler la routine (p2 et x sur la vue 56). En Eiffel, il n'existe qu'un seul mode de passage des arguments : **le mode de passage par valeur**. Ce mode correspond au mode de passage sans le mot clé **var** du langage Pascal ou bien au mode de passage par défaut du langage C. Il consiste à initialiser les arguments formels avec une copie de la valeur des arguments effectifs. La place mémoire pour les arguments formels est allouée dans la pile (comme pour les variables locales). Si un argument correspond à une classe **expanded**, on passe dans la pile une copie de la valeur correspondante. Si un argument est un pointeur vers une instance (classe non **expanded**), on passe dans la pile une copie de ce pointeur. Le pointeur sur le receveur, **Current**, est également passé dans la pile ainsi que l'adresse du point de retour vers l'appelant. La phase de liaison entre arguments formels et arguments effectifs ne provoque pas d'instanciation<sup>1</sup>.

Sur l'exemple, on suppose qu'il existe une routine **selecteur** dans la classe POINT, et l'on considère un appel de cette routine (vue 56). La vue 57 montre l'état de la pile avant l'appel de la routine **selecteur** ainsi que l'état de la pile au début de l'exécution de la routine **selecteur**, juste après la phase de liaison entre les arguments formels et les arguments effectifs. L'association effectuée entre arguments formels et arguments effectifs montre que la routine appelée est susceptible de modifier les arguments manipulés par des pointeurs. Les instances pointées avant l'appel par p1 et p2 peuvent être modifiées par l'exécution de la routine **selecteur**. Le réel x ne peut pas être modifié par la routine **selecteur**.

```
p1, p2 : POINT
x : REAL
...
-- ⊙ Avant appel
p1.selecteur(p2,x)
-- ⊙ Après appel
```

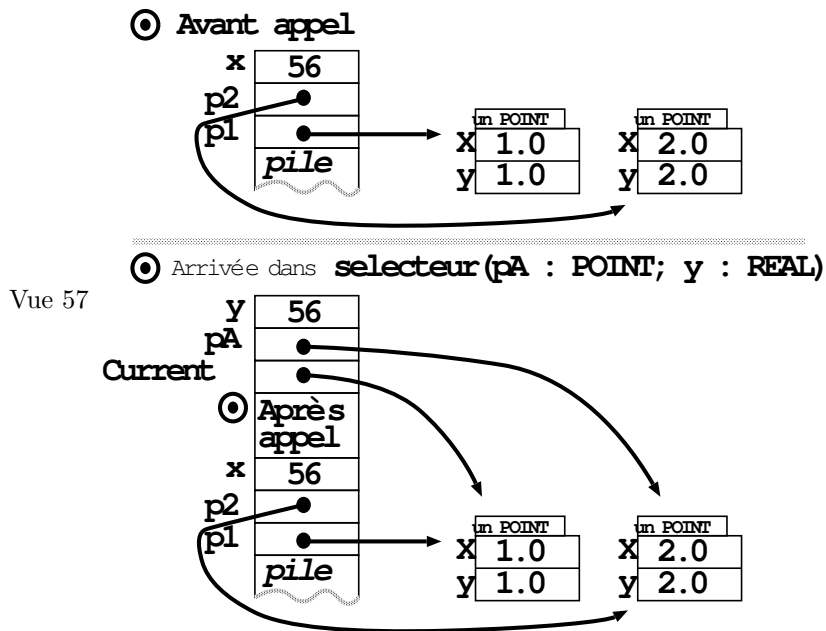
Vue 56

### arguments des routines

```
class POINT
...
selecteur(pA : POINT; y : REAL)
...
end -- class POINT
```

---

<sup>1</sup>Sauf si l'on considère le passage d'un type **expanded** dans type référence compatible. Encore une fois, comme la manipulation des classes **expanded** est à mon avis une affaire de spécialistes, je préfère laisser ce cas particulier de côté pour l'instant.



## 7 ARRAY et STRING

Ces classes, bien que traitées de façon spécifique par le compilateur Eiffel, sont décrites complètement en Eiffel (cf. `.../SmartEiffel/lib/kernel/array.e` pour la classe `ARRAY` et, dans le même répertoire, `string.e` pour la classe `STRING`). En fait, le seul traitement spécifique effectué par le compilateur pour ces deux classes consiste à prendre en compte, d'une part la notation des chaînes de caractère explicites (comme "Hello World." par exemple) ainsi que d'autre part la notation de construction explicite de tableaux (vue 63).

### 7.1 Classe `STRING`

Les chaînes de caractères sont représentées habituellement à l'aide d'objets de la classe `STRING`. Ces chaînes de caractères sont de longueur variable et peuvent s'allonger ou rétrécir durant l'exécution, selon les besoins du programmeur.

La vue 58 présente quelques une des routines de la classe `STRING`. Cette classe offre aussi la possibilité d'étendre une chaîne de caractères existante (vue 59, procédure `extend` et `precede`). Il est important de constater que, dans le cas où deux variables référencent (pointent) la même chaîne, et lorsque l'on étend la chaîne en question, les deux variables pointent toujours la même chaîne (physique). Ceci donne alors l'impression que les deux chaînes ont été étendues simultanément alors qu'en fait il n'existe qu'une seule et unique chaîne en mémoire. Dans le cas où le même et unique objet est accessible par deux chemins différents, on parle souvent d'*aliasing*. Ceci se produit très souvent et il est nécessaire de bien maîtriser ces phénomènes d'*aliasing* pour écrire des application efficaces et moins gourmandes en mémoire.

Si l'on consulte le texte source de la classe `STRING` on découvre comment il est possible de changer dynamiquement la taille d'une chaîne mémorisée, sans changer l'emplacement mémoire de l'instance initiale (vue 60). Partant d'une variable de type `STRING`, il faut suivre deux pointeurs pour atteindre un caractère particulier de la chaîne. L'instance de la classe `NATIVE_ARRAY` correspond à une chaîne de caractères ordinaire (une chaîne de caractères du langage C), non extensible et rangée de façon contiguë en mémoire. En cas d'extension de la chaîne de la classe `STRING`, on alloue en mémoire une nouvelle instance de la classe `NATIVE_ARRAY`. L'instance de la classe `STRING` ne change pas d'adresse en mémoire.

Pour couronner le tout, la classe `STRING` possède une définition **spécifique** de la procédure `copy` qui copie aussi la zone de stockage des caractères (vue 61). La fonction `copy` de la classe `STRING` permet donc d'obtenir une copie de la chaîne initiale sans aucune zone mémoire commune avec cette chaîne initiale.

### Client de la classe `STRING`

```

s1, s2 : STRING
c : CHARACTER
...
-- Instancier une chaîne de 3 caractères espace :
create s1.make(3)
Vue 58 -- Changer le premier caractère :
s1.put('a',1)
-- Changer le troisième caractère :
s1.put('z',s1.count)
-- Deux notations (équivalentes) pour consulter :
c := s1.item(2)
c := s1 @ 2
-- Instancier une sous-chaîne :
s2 := s1.substring(2,3)

```



```

-- Étendre à droite :
s2.extend('x')
-- Étendre à gauche :
s2.precede(c)
vrai := ((s2.item(s2.count)) = 'x') -- ← True
    
```

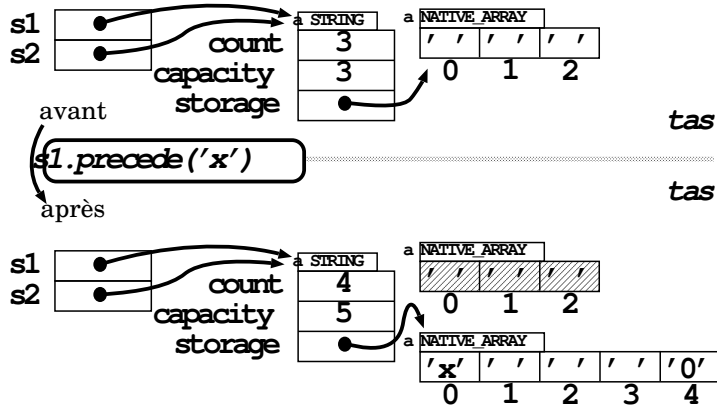
Vue 59

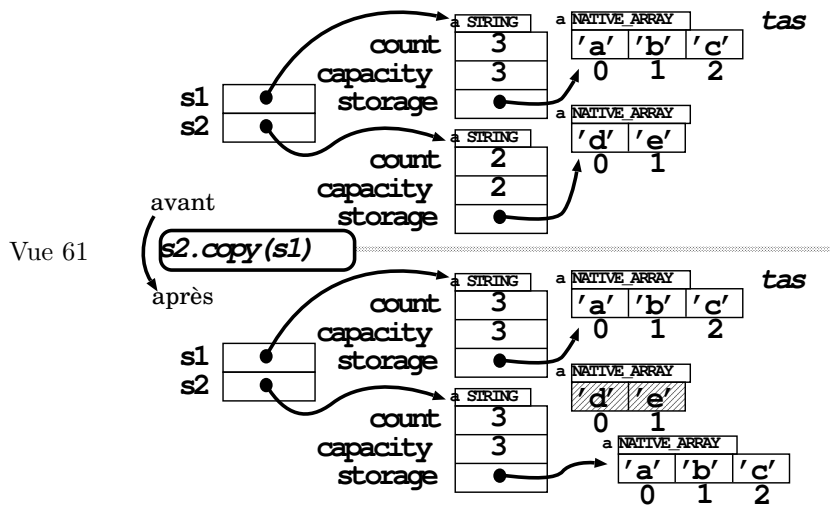
### Comment font ils ça ?

```

create s1.make_filled(' ', 3)
s2 := s1
s1.precede('x')
vrai1 := ((s2.item(1)) = 'x') -- ← True
vrai2 := (s1 = s2) -- ← True
    
```

Vue 60





## 7.2 Interface de la classe STRING – short

```
class interface STRING
--
-- Resizable character STRINGS indexed from 1 to count.
--
creation
make (needed_capacity: INTEGER)
-- Initialize the string to have at least needed_capacity
-- characters of storage.
require
non_negative_size: needed_capacity >= 0
ensure
needed_capacity <= capacity;
empty_string: count = 0
copy (other: STRING)
-- Copy other onto Current.
require
other_not_void: other /= Void
ensure
count = other.count;
is_equal: is_equal(other)
make_empty
-- Create an empty string.
make_filled (c: CHARACTER; n: INTEGER)
-- Initialize string with n copies of c.
require
```

```

        valid_count: n >= 0
    ensure
        count_set: count = n;
        filled: occurrences(c) = count
from_external (p: POINTER)
    -- Internal storage is set using p (may be dangerous because
    -- the external C string p is not duplicated).
    -- Assume p has a null character at the end in order to
    -- compute the Eiffel count. This extra null character
    -- is not part of the Eiffel STRING.
    -- Also consider from_external_copy to choose the most appropriate.
    require
        p.is_not_null
    ensure
        capacity = count + 1;
        p = to_external
from_external_copy (p: POINTER)
    -- Internal storage is set using a copy of p.
    -- Assume p has a null character at the end in order to
    -- compute the Eiffel count. This extra null character
    -- is not part of the Eiffel STRING.
    -- Also consider from_external to choose the most appropriate.
    require
        p.is_not_null
feature(s)
    hash_code: INTEGER
        -- The hash-code value of Current.
    ensure
        good_hash_value: Result >= 0
    is_equal (other: STRING): BOOLEAN
        -- Do both strings have the same character sequence?
        -- (Redefined from GENERAL)
    require
        other_not_void: other /= Void
    ensure
        trichotomy: Result = (not (Current < other) and not (other < Current));
        consistent: standard_is_equal(other) implies Result;
        symmetric: Result implies other.is_equal(Current)
    infix "<" (other: STRING): BOOLEAN
        -- Is Current less than other?
    require
        other_exists: other /= Void
    ensure
        asymmetric: Result implies not (other < Current)
    infix "<=" (other: STRING): BOOLEAN
        -- Is Current less than or equal other?
    require
        other_exists: other /= Void
    ensure
        definition: Result = (Current < other or is_equal(other))
    infix ">" (other: STRING): BOOLEAN
        -- Is Current strictly greater than other?
    require
        other_exists: other /= Void
    ensure
        definition: Result = (other < Current)
    infix ">=" (other: STRING): BOOLEAN

```

```

    -- Is Current greater than or equal than other?
    require
        other_exists: other /= Void
    ensure
        definition: Result = (other <= Current)
in_range (lower, upper: STRING): BOOLEAN
    -- Return true if Current is in range [lower..upper]
    ensure
        Result = (Current >= lower and Current <= upper)
compare (other: STRING): INTEGER
    -- If current object equal to other, 0;
    -- if smaller, -1; if greater, 1.
    require
        other_exists: other /= Void
    ensure
        equal_zero: Result = 0 = is_equal(other);
        smaller_negative: Result = - 1 = Current < other;
        greater_positive: Result = 1 = Current > other
three_way_comparison (other: STRING): INTEGER
    -- If current object equal to other, 0;
    -- if smaller, -1; if greater, 1.
    require
        other_exists: other /= Void
    ensure
        equal_zero: Result = 0 = is_equal(other);
        smaller_negative: Result = - 1 = Current < other;
        greater_positive: Result = 1 = Current > other
min (other: STRING): STRING
    -- Minimum of Current and other.
    require
        other /= Void
    ensure
        Result <= Current and then Result <= other;
        compare(Result) = 0 or else other.compare(Result) = 0
max (other: STRING): STRING
    -- Maximum of Current and other.
    require
        other /= Void
    ensure
        Result >= Current and then Result >= other;
        compare(Result) = 0 or else other.compare(Result) = 0
count: INTEGER
    -- String length which is also the maximum valid index.
capacity: INTEGER
    -- Capacity of the storage area.
lower: INTEGER
    -- Minimum index; actually, this is always 1 (this feature is
    -- here to mimic the one of the COLLECTIONs hierarchy).
upper: INTEGER
    -- Maximum index; actually the same value as count (this
    -- feature is here to mimic the one of the COLLECTION hierarchy).
    ensure
        Result = count
feature(s) -- Creation / Modification:
make (needed_capacity: INTEGER)
    -- Initialize the string to have at least needed_capacity
    -- characters of storage.

```

```

    require
        non_negative_size: needed_capacity >= 0
    ensure
        needed_capacity <= capacity;
        empty_string: count = 0
make_empty
    -- Create an empty string.
make_filled (c: CHARACTER; n: INTEGER)
    -- Initialize string with n copies of c.
    require
        valid_count: n >= 0
    ensure
        count_set: count = n;
        filled: occurrences(c) = count
feature(s) -- Testing:
is_empty: BOOLEAN
    -- Has string length 0?
item (i: INTEGER): CHARACTER
    -- Character at position i.
    require
        valid_index: valid_index(i)
valid_index (i: INTEGER): BOOLEAN
    -- True when i is valid (i.e., inside actual bounds).
    ensure
        definition: Result = (1 <= i and i <= count)
same_as (other: STRING): BOOLEAN
    -- Case insensitive is_equal.
    require
        other /= Void
item_code (i: INTEGER): INTEGER
    -- Code of character at position i.
    require
        valid_index: valid_index(i)
index_of (c: CHARACTER; start_index: INTEGER): INTEGER
    -- Index of first occurrence of c at or after start_index,
    -- 0 if none.
    --
    -- Note: see also first_index_of to start searching at 1.
    -- Actually first_index_of is not exactly the equivalent of index_of
    -- in release -0.76: when the search failed the result is
    -- now 0 (and no longer count + 1). So, to update your code from
    -- release -0.76 to release -0.75, replace index_of with first_index_of
    -- and be careful to see what's done with the result !
    require
        valid_start_index: start_index >= 1 and start_index <= count + 1
    ensure
        Result /= 0 implies item(Result) = c
first_index_of (c: CHARACTER): INTEGER
    -- Index of first occurrence of c at index 1 or after index 1.
    ensure
        definition: Result = index_of(c,1)
has (c: CHARACTER): BOOLEAN
    -- True if c is in the STRING.
has_substring (other: STRING): BOOLEAN
    -- True if Current contains other.
    require
        other_not_void: other /= Void

```

```

occurrences (c: CHARACTER): INTEGER
  -- Number of times character c appears in the string.
  ensure
    Result >= 0
has_suffix (s: STRING): BOOLEAN
  -- True if suffix of Current is s.
  require
    s /= Void
has_prefix (p: STRING): BOOLEAN
  -- True if prefix of Current is p.
  require
    p /= Void
feature(s) -- Testing and Conversion:
is_boolean: BOOLEAN
  -- Does Current represent a BOOLEAN?
  -- Valid BOOLEANS are "true" and "false".
to_boolean: BOOLEAN
  -- Boolean value;
  -- "true" yields true, "false" yields false (what a surprise).
  require
    represents_a_boolean: is_boolean
is_integer: BOOLEAN
  -- Does 'Current' represent an INTEGER?
  -- Result is true if and only if the following two conditions hold:
  --
  -- 1. In the following BNF grammar, the value of Current can be
  -- produced by "Integer_literal", if leading and trailing
  -- separators are ignored:
  --
  -- Integer_literal = [Sign] Integer
  -- Sign           = "+" — "-"
  -- Integer        = Digit — Digit Integer
  -- Digit          = "0" — "1" — "2" — "3" — "4" — "5" — "6" — "7" — "8" — "9"
  --
  -- 2. The numerical value represented by Current is within the
  -- range that can be represented by an instance of type INTEGER.
to_integer: INTEGER
  -- Current must look like an INTEGER.
  require
    is_integer
is_double: BOOLEAN
  -- Can contents be read as a DOUBLE?
  -- Fails for numbers where the base or "10 ^ exponent" are not in
  -- the range Minimum_double ... Maximum_double. Parsing is done
  -- positive. That means if Minimum_double.abs is not equal to
  -- Maximum_double it will not work correctly. Furthermore the
  -- arithmetic package used must support the value 'inf' for a
  -- number greater than Maximum_double.
  -- Result is true if and only if the following two conditions hold:
  --
  -- 1. In the following BNF grammar, the value of Current can be
  -- produced by "Real_literal", if leading or trailing separators
  -- are ignored.
  --
  -- Real_literal   = Mantissa [Exponent-part]
  -- Exponent-part = "E" Exponent
  --              — "e" Exponent

```

```

-- Exponent      = Integer_literal
-- Mantissa      = Decimal_literal
-- Decimal_literal = Integer_literal [ "." Integer ]
-- Integer_literal = [ Sign ] Integer
-- Sign          = "+" - "-"
-- Integer       = Digit - Digit Integer
-- Digit         = "0" - "1" - "2" - "3" - "4" - "5" - "6" - "7" - "8" - "9"
--
--
-- 2. The numerical value represented by Current is within the range
-- that can be represented by an instance of type DOUBLE.
to_double: DOUBLE
-- Conversion to the corresponding DOUBLE value. The string must
-- look like a DOUBLE (or like an INTEGER because fractionnal part
-- is optional). For an exact definition see 'is_double'.
-- Note that this conversion might not be exact.
require
  represents_a_double: is_double
is_real: BOOLEAN
-- Can contents be read as a REAL?
to_real: REAL
-- Conversion to the corresponding REAL value.
-- The string must look like a REAL (or like an
-- INTEGER because fractionnal part is optional).
require
  is_integer or is_real
is_number: BOOLEAN
-- Can contents be read as a NUMBER?
to_number: NUMBER
-- Current must look like an INTEGER.
require
  is_number
is_bit: BOOLEAN
-- True when the contents is a sequence of bits (i.e., mixed
-- characters 0 and characters 1).
ensure
  Result = (count = occurrences('0') + occurrences('1'))
to_hexadecimal
-- Convert Current bit sequence into the corresponding
-- hexadecimal notation.
require
  is_bit
binary_to_integer: INTEGER
-- Assume there is enough space in the INTEGER to store
-- the corresponding decimal value.
require
  is_bit;
  count <= Integer_bits
feature(s) -- Modification:
resize (new_count: INTEGER)
-- Resize Current. When new_count is greater than
-- count, new positions are initialized with the
-- default value of type CHARACTER ('%U').
require
  new_count >= 0
ensure
  count = new_count;

```

```

        capacity >= old capacity
clear
  -- Clear out the current STRING.
  -- Note: internal storage memory is neither released nor shrunk.
  ensure
    count = 0
wipe_out
  -- Clear out the current STRING.
  -- Note: internal storage memory is neither released nor shrunk.
  ensure
    count = 0
copy (other: STRING)
  -- Copy other onto Current.
  require
    other_not_void: other /= Void
  ensure
    count = other.count;
    is_equal: is_equal(other)
fill_with (c: CHARACTER)
  -- Replace every character with c.
  ensure
    occurrences(c) = count
replace_all (old_character, new_character: CHARACTER)
  -- Replace all occurrences of the element old_character by
  -- new_character.
  ensure
    count = old count;
    occurrences(old_character) = 0
append (s: STRING)
  -- Append a copy of 's' to Current.
  require
    s_not_void: s /= Void
append_string (s: STRING)
  -- Append a copy of 's' to Current.
  require
    s_not_void: s /= Void
prepend (other: STRING)
  -- Prepend other to Current.
  require
    other /= Void
  ensure
    Current.is_equal(old other.twin + old Current.twin)
insert_string (s: STRING; i: INTEGER)
  -- Insert s at index i, shifting characters from index i
  -- to count rightwards.
  require
    string_not_void: s /= Void;
    valid_insertion_index: 1 <= i and i <= count + 1
replace_substring (s: STRING; start_index, end_index: INTEGER)
  -- Replace the substring from start_index to end_index,
  -- inclusive, with s.
  require
    string_not_void: s /= Void;
    valid_start_index: 1 <= start_index;
    valid_end_index: end_index <= count;
    meaningful_interval: start_index <= end_index + 1
infix "+" (other: STRING): STRING

```



```

-- Create a new STRING which is the concatenation of
-- Current and other.
require
  other_exists: other /= Void
ensure
  result_count: Result.count = count + other.count
put (c: CHARACTER; i: INTEGER)
-- Put c at index i.
require
  valid_index: valid_index(i)
ensure
  item(i) = c
swap (i1, i2: INTEGER)
require
  valid_index(i1);
  valid_index(i2)
ensure
  item(i1) = old item(i2);
  item(i2) = old item(i1)
insert_character (c: CHARACTER; i: INTEGER)
-- Inserts c at index i, shifting characters from
-- position 'i' to count rightwards.
require
  valid_insertion_index: 1 <= i and i <= count + 1
ensure
  item(i) = c
shrink (min_index, max_index: INTEGER)
-- Keep only the slice [min_index .. max_index] or nothing
-- when the slice is empty.
require
  1 <= min_index;
  max_index <= count;
  min_index <= max_index + 1
ensure
  count = max_index - min_index + 1
remove (i: INTEGER)
-- Remove character at position i.
require
  valid_removal_index: valid_index(i)
ensure
  count = old count - 1
add_first (c: CHARACTER)
-- Add c at first position.
ensure
  count = 1 + old count;
  item(1) = c
precede (c: CHARACTER)
-- Add c at first position.
ensure
  count = 1 + old count;
  item(1) = c
add_last (c: CHARACTER)
-- Append c to string.
ensure
  count = 1 + old count;
  item(count) = c
append_character (c: CHARACTER)

```

```

    -- Append c to string.
    ensure
        count = 1 + old count;
        item(count) = c
extend (c: CHARACTER)
    -- Append c to string.
    ensure
        count = 1 + old count;
        item(count) = c
to_lower
    -- Convert all characters to lower case.
to_upper
    -- Convert all characters to upper case.
as_lower: STRING
    -- New object with all letters in lower case.
as_upper: STRING
    -- New object with all letters in upper case.
keep_head (n: INTEGER)
    -- Remove all characters except for the first n.
    -- Do nothing if n  $\leq$  count.
    require
        n_non_negative: n  $\geq$  0
    ensure
        count = n.min(old count)
keep_tail (n: INTEGER)
    -- Remove all characters except for the last n.
    -- Do nothing if n  $\leq$  count.
    require
        n_non_negative: n  $\geq$  0
    ensure
        count = n.min(old count)
remove_head (n: INTEGER)
    -- Remove n first characters.
    -- If n  $\leq$  count, remove all.
    require
        n_non_negative: n  $\geq$  0
    ensure
        count = (0).max(old count - n)
remove_first (n: INTEGER)
    -- Remove n first characters.
    -- If n  $\leq$  count, remove all.
    require
        n_non_negative: n  $\geq$  0
    ensure
        count = (0).max(old count - n)
remove_tail (n: INTEGER)
    -- Remove n last characters.
    -- If n  $\leq$  count, remove all.
    require
        n_non_negative: n  $\geq$  0
    ensure
        count = (0).max(old count - n)
remove_last (n: INTEGER)
    -- Remove n last characters.
    -- If n  $\leq$  count, remove all.
    require
        n_non_negative: n  $\geq$  0

```

```

    ensure
      count = (0).max(old count - n)
remove_substring (start_index, end_index: INTEGER)
  -- Remove all characters from strt_index to end_index inclusive.
  require
    valid_start_index: 1 <= start_index;
    valid_end_index: end_index <= count;
    meaningful_interval: start_index <= end_index + 1
  ensure
    count = old count - end_index - start_index + 1
remove_between (start_index, end_index: INTEGER)
  -- Remove all characters from strt_index to end_index inclusive.
  require
    valid_start_index: 1 <= start_index;
    valid_end_index: end_index <= count;
    meaningful_interval: start_index <= end_index + 1
  ensure
    count = old count - end_index - start_index + 1
remove_suffix (s: STRING)
  -- Remove the suffix s of current string.
  require
    has_suffix(s)
  ensure
    (old Current.twin).is_equal(Current + old s.twin)
remove_prefix (s: STRING)
  -- Remove the prefix s of current string.
  require
    has_prefix(s)
  ensure
    (old Current.twin).is_equal(old s.twin + Current)
left_adjust
  -- Remove leading blanks.
  ensure
    stripped: is_empty or else item(1) /= ' '
right_adjust
  -- Remove trailing blanks.
  ensure
    stripped: is_empty or else item(count) /= ' '
feature(s) -- Printing:
  out_in_tagged_out_memory
    -- Append terse printable representation of current object
    -- in tagged_out_memory.
  fill_tagged_out_memory
    -- Append a viewable information in tagged_out_memory in
    -- order to affect the behavior of out, tagged_out, etc.
feature(s) -- Other features:
  first: CHARACTER
    -- Access to the very first character.
    require
      not is_empty
    ensure
      definition: Result = item(1)
  last: CHARACTER
    -- Access to the very last character.
    require
      not is_empty
    ensure

```

```

        definition: Result = item(count)
substring (start_index, end_index: INTEGER): STRING
-- New string consisting of items [start_index.. end_index].
require
    valid_start_index: 1 <= start_index;
    valid_end_index: end_index <= count;
    meaningful_interval: start_index <= end_index + 1
ensure
    substring_count: Result.count = end_index - start_index + 1
extend_multiple (c: CHARACTER; n: INTEGER)
-- Extend Current with n times character c.
require
    n >= 0
ensure
    count = n + old count
precede_multiple (c: CHARACTER; n: INTEGER)
-- Prepend n times character c to Current.
require
    n >= 0
ensure
    count = n + old count
extend_to_count (c: CHARACTER; needed_count: INTEGER)
-- Extend Current with c until needed_count is reached.
-- Do nothing if needed_count is already greater or equal
-- to count.
require
    needed_count >= 0
ensure
    count >= needed_count
precede_to_count (c: CHARACTER; needed_count: INTEGER)
-- Prepend c to Current until needed_count is reached.
-- Do nothing if needed_count is already greater or equal
-- to count.
require
    needed_count >= 0
ensure
    count >= needed_count
reverse
-- Reverse the string.
remove_all_occurrences (ch: CHARACTER)
-- Remove all occurrences of ch.
ensure
    count = old count - old occurrences(ch)
substring_index (other: STRING; start_index: INTEGER): INTEGER
-- Position of first occurrence of other at or after start;
-- 0 if none.
require
    other_not_void: other /= Void;
    valid_start_index: start_index >= 1 and start_index <= count + 1
first_substring_index (other: STRING): INTEGER
-- Position of first occurrence of other at or after 1;
-- 0 if none.
require
    other_not_void: other /= Void
ensure
    definition: Result = substring_index(other,1)
feature(s) -- Splitting a STRING:

```

```

split: ARRAY[STRING]
  -- Split the string into an array of words. Uses is_separator of
  -- CHARACTER to find words. Gives Void or a non empty array.
  ensure
    Result /= Void implies not Result.is_empty
split_in (words: COLLECTION[STRING])
  -- Same jobs as split but result is appended in words.
  require
    words /= Void
  ensure
    words.count >= old words.count
feature(s) -- Other features:
extend_unless (ch: CHARACTER)
  -- Extend Current (using extend) with ch unless ch is
  -- already the last character.
  ensure
    last = ch;
    count >= old count
get_new_iterator: ITERATOR[CHARACTER]
feature(s) -- Interfacing with C string:
to_external: POINTER
  -- Gives C access to the internal storage (may be dangerous).
  -- To be compatible with C, a null character is added at the end
  -- of the internal storage. This extra null character is not
  -- part of the Eiffel STRING.
  ensure
    count = old count;
    Result.is_not_null
from_external (p: POINTER)
  -- Internal storage is set using p (may be dangerous because
  -- the external C string p is not duplicated).
  -- Assume p has a null character at the end in order to
  -- compute the Eiffel count. This extra null character
  -- is not part of the Eiffel STRING.
  -- Also consider from_external_copy to choose the most appropriate.
  require
    p.is_not_null
  ensure
    capacity = count + 1;
    p = to_external
from_external_copy (p: POINTER)
  -- Internal storage is set using a copy of p.
  -- Assume p has a null character at the end in order to
  -- compute the Eiffel count. This extra null character
  -- is not part of the Eiffel STRING.
  -- Also consider from_external to choose the most appropriate.
  require
    p.is_not_null
from_c (p: POINTER)
  -- Internal storage is set using a copy of p.
  -- Assume p has a null character at the end in order to
  -- compute the Eiffel count. This extra null character
  -- is not part of the Eiffel STRING.
  -- Also consider from_external to choose the most appropriate.
  require
    p.is_not_null
invariant

```

```

0 <= count;
count <= capacity;
capacity > 0 implies storage.is_not_null;
end of STRING

```

### 7.3 Classe ARRAY

La classe ARRAY correspond à la notion de vecteur ; les instances de cette classe sont des tableaux à une seule dimension (vue 62). Ces tableaux sont extensibles (agrandissement et/ou rétrécissement) et l'indexation pour accéder aux éléments est quelconque (l'indice du premier élément n'est pas forcément 1).

Lors de la déclaration d'un tableau, il faut indiquer le type des éléments que l'on souhaite ranger dans le tableau. Comme pour toutes les classes ordinaires, la déclaration d'une variable de type ARRAY[INTEGER] ne provoque que la réservation d'une place mémoire pour un pointeur sur le futur tableau que l'on souhaite manipuler. Pour obtenir effectivement le tableau, il faut comme d'habitude l'instancier. Notons que l'on doit préciser au moment de l'instanciation la tranche des index souhaités (bornes incluses).

Pour le contenu du tableau, la distinction habituelle entre les types de base et les vraies classes s'applique aussi. Par exemple, si on instancie un tableau de quatre points (de la classe POINT), aucun point de la classe POINT n'est créé. Seule la place pour quatre pointeurs est allouée. Inversement, si on instancie un tableau de quatre entiers (de la classe INTEGER), la place pour les quatre entiers est allouée.

L'interface de la classe ARRAY possède de nombreux points communs avec l'interface de la classe STRING. En particulier, les routines d'accès en lecture et en écriture ont le même profil. Comme les chaînes de la classe STRING, les tableaux de la classe ARRAY sont extensibles (vue 63), et comme dans la classe STRING, il faut suivre plusieurs pointeurs pour atteindre un élément du tableau.

Une notation spéciale permet d'instancier et d'initialiser de manière plus concise un tableau dont les indices commencent à l'index 1. Cette notation est présentée sur la vue 63. Du point de vue du code machine généré, c'est exactement la même chose qu'une instanciation d'un nouveau tableau suivi d'une suite d'appels de la procédure put (vue 64).

La bibliothèque de classes de SmartEiffel comporte également deux autres catégories de vecteurs (tableaux) :

- les vecteurs de la classe FAST\_ARRAY qui ont globalement la même interface que ARRAY sachant que l'index du premier élément est fixé par la constante 0 ; ces tableaux également redimensionables permettent parfois d'obtenir de meilleures performances à l'exécution ;
- les tableaux primitifs de la classe NATIVE\_ARRAY ne sont pas redimensionables et l'index du premier élément est également fixé à 0 ; l'usage de ces tableaux primitifs permet d'obtenir les mêmes performances qu'avec les tableaux (très primitifs eux aussi) du langage C ; l'usage de ces tableaux est délicate et est habituellement réservée aux experts (voir par exemple l'utilisation de la classe NATIVE\_ARRAY pour l'implantation de la classe STRING sur la vue 60 et 61).

Enfin, sans trop anticiper car nous aurons l'occasion d'y revenir, la bibliothèque de SmartEiffel comporte également d'autres structures de données permettant la manipulation de collections d'objets :

- la classe LINKED\_LIST pour les listes simplement chaînées,
- la classe TWO\_WAY\_LINKED\_LIST pour les listes doublement chaînées,
- la classe RING\_ARRAY pour les files d'attente,
- la classe SET pour les ensembles, sachant qu'il existe une implantation à base de hachage nommée HASHED\_SET et une seconde implantation à base d'arbres binaires AVL nommée AVL\_SET,

- la classe `DICTIONARY` pour les tables ; sachant qu'il existe une implantation à base de hachage nommée `HASHED_DICTIONARY` et une seconde implantation à base d'arbres binaires AVL nommée `AVL_DICTIONARY`,
- etc.

**Exercice 13** Est-il possible d'écrire de façon plus concise le texte Eiffel de la vue 62 ?

Comme la classe `STRING`, la classe `ARRAY` possède une version spécifique de la procédure `copy`. Le choix effectué pour la classe `ARRAY` est cohérent : en cas de copie d'un tableau, les éléments pointés ne sont pas dupliqués. Par exemple, si l'on effectue la copie d'un tableau de points, aucune nouvelle instance de la classe `POINT` n'est instanciée. Les mêmes instances sont pointées par les deux tableaux. Bien entendu, si on souhaite dupliquer aussi les points, il faut utiliser `deep_twin` à la place de `copy`. Attention au gaspillage ! Choisissez judicieusement et justifiez la formule la mieux adaptée.

**Exercice 14** Comment faut-il définir la classe `C` afin que toutes les instances de la classe `C` puissent accéder à un unique et même tableau d'entiers ? Ce tableau d'entiers ne doit être accessible qu'aux instances de la classe `C`.

### Client de la classe `ARRAY`

```

t1 : ARRAY[POINT]
pt : POINT
...
...
Vue 62  -- Instanciation d'un tableau de trois
        -- pointeurs de points :
create t1.make(2,4)
create pt.make(1,1)
t1.put(pt,2)
t1.put(pt,3)
pt := t1.item(4)

```

**ARRAY : tableaux extensibles**

```
-- Étend si besoin le tableau, puis effectue
-- l'équivalent d'un put:
t1.force(pt,5)
```

Vue 63

**Instanciation**

```
t2 : ARRAY[CHARACTER]
...
t2 := <<'a','b','c'>>
vrai1 := ((t2.first) = 'a') -- ← True
vrai2 := ((t2.upper) = 3) -- ← True
```

```
tabPoint : ARRAY[POINT]
pt : POINT
tr : TRIANGLE
```

---

```
tabPoint := <<pt,tr.p1,tr.p2,Void>>
```

Vue 64

⇔

```
create tabPoint.make(1,4)
tabPoint.put(pt,1)
tabPoint.put(tr.p1,2)
tabPoint.put(tr.p2,3)
tabPoint.put(Void,4)
```



## 7.4 Interface de la classe ARRAY

```

class interface ARRAY[E]
  --
  -- General purpose resizable ARRAYS.
  --
  creation
    make (min_index, max_index: INTEGER)
      -- Prepare the array to hold values for indexes in range
      -- [min_index .. max_index]. Set all values to default.
      -- When max_index = min_index - 1, the array is empty.
      require
        valid_bounds: min_index <= max_index + 1
      ensure
        lower_set: lower = min_index;
        upper_set: upper = max_index;
        items_set: all_default
    with_capacity (needed_capacity, low: INTEGER)
      -- Create an empty array with capacity initialized
      -- at least to needed_capacity and lower set to low.
      require
        needed_capacity >= 0
      ensure
        is_empty;
        needed_capacity <= capacity;
        lower = low
    from_collection (model: COLLECTION[E])
      -- Initialize the current object with the contents of model.
      require
        model /= Void
      ensure
        lower = model.lower;
        upper = model.upper;
        count = model.count
  feature(s) -- Indexing:
    lower: INTEGER
      -- Lower index bound.
    upper: INTEGER
      -- Upper index bound.
    valid_index (index: INTEGER): BOOLEAN
      -- True when index is valid (ie. inside actual
      -- bounds of the collection).
      ensure
        Result = (lower <= index and then index <= upper)
  feature(s) -- Counting:
    count: INTEGER
      -- Number of available indices.
      ensure
        Result = upper - lower + 1
    is_empty: BOOLEAN
      -- Is collection empty ?
      ensure
        Result = (count = 0)
  feature(s) -- Accessing:
    item (i: INTEGER): E
      -- Item at the corresponding index i.
      require
        valid_index(i)

```

```

first: E
  -- The very first item.
  require
    count >= 1
  ensure
    Result = item(lower)
last: E
  -- The last item.
  require
    not is.empty
  ensure
    Result = item(upper)
feature(s) -- Writing:
  put (element: E; i: INTEGER)
    -- Make element the item at index i.
    require
      valid_index(i)
    ensure
      item(i) = element;
      count = old count
  swap (i1, i2: INTEGER)
    -- Swap item at index i1 with item at index i2.
    require
      valid_index(i1);
      valid_index(i2)
    ensure
      item(i1) = old item(i2);
      item(i2) = old item(i1);
      count = old count
  set_all_with (v: E)
    -- Set all items with value v.
    ensure
      count = old count
  set_slice_with (v: E; lower_index, upper_index: INTEGER)
    -- Set all items in range [lower_index .. upper_index] with v.
    require
      lower_index <= upper_index;
      valid_index(lower_index);
      valid_index(upper_index)
    ensure
      count = old count
  clear_all
    -- Set every item to its default value.
    -- The count is not affected (see also clear).
    ensure
      stable_upper: upper = old upper;
      stable_lower: lower = old lower;
      all_default
feature(s) -- Adding:
  add_first (element: E)
    -- Add a new item in first position : count is increased by
    -- one and all other items are shifted right.
    ensure
      first = element;
      count = 1 + old count;
      lower = old lower;
      upper = 1 + old upper

```

```

add_last (element: E)
  -- Add a new item at the end : count is increased by one.
  ensure
    last = element;
    count = 1 + old count;
    lower = old lower;
    upper = 1 + old upper
add (element: E; index: INTEGER)
  -- Add a new element at rank index : count is increased
  -- by one and range [index .. upper] is shifted right
  -- by one position.
  require
    index.in_range(lower, upper + 1)
  ensure
    item(index) = element;
    count = 1 + old count;
    upper = 1 + old upper
append_collection (other: COLLECTION[E])
  -- Append other to Current.
  require
    other /= Void
  ensure
    count = other.count + old count
feature(s) -- Modification:
force (element: E; index: INTEGER)
  -- Make element the item at index, enlarging the collection if
  -- necessary (new bounds except index are initialized with
  -- default values).
  require
    true
  require else
    index >= lower
  ensure
    lower = index.min(old lower);
    upper = index.max(old upper);
    item(index) = element
copy (other: ARRAY[E])
  -- Reinitialize by copying all the items of other.
  require
    other_not_void: other /= Void
  ensure
    is_equal: is_equal(other)
from_collection (model: COLLECTION[E])
  -- Initialize the current object with the contents of model.
  require
    model /= Void
  ensure
    lower = model.lower;
    upper = model.upper;
    count = model.count
feature(s) -- Removing:
remove_first
  -- Remove the first element of the collection.
  require
    not is_empty
  ensure
    upper = old upper;

```

```

        count = old count - 1;
        lower = old lower + 1 xor upper = old upper - 1
remove (index: INTEGER)
    -- Remove the item at position index. Followings items
    -- are shifted left by one position.
    require
        valid_index(index)
    ensure
        count = old count - 1;
        upper = old upper - 1
remove_last
    -- Remove the last item.
    require
        not is_empty
    ensure
        count = old count - 1;
        upper = old upper - 1
clear
    -- Discard all items in order to make it is_empty.
    -- See also clear_all.
    ensure
        capacity = old capacity;
        is_empty
feature(s) -- Looking and Searching:
has (x: E): BOOLEAN
    -- Look for x using equal for comparison.
    -- Also consider fast_has to choose the most appropriate.
fast_has (x: E): BOOLEAN
    -- Look for x using basic = for comparison.
    -- Also consider has to choose the most appropriate.
index_of (element: E): INTEGER
    -- Give the index of the first occurrence of element using
    -- is_equal for comparison.
    -- Answer upper + 1 when element is not inside.
    -- Also consider fast_index_of to choose the most appropriate.
    ensure
        lower <= Result;
        Result <= upper + 1;
        Result <= upper implies equal(element,item(Result))
fast_index_of (element: E): INTEGER
    -- Give the index of the first occurrence of element using
    -- basic = for comparison.
    -- Answer upper + 1 when element is not inside.
    -- Also consider index_of to choose the most appropriate.
    ensure
        lower <= Result;
        Result <= upper + 1;
        Result <= upper implies element = item(Result)
feature(s) -- Looking and comparison:
is_equal (other: ARRAY[E]): BOOLEAN
    -- Do both collections have the same lower, upper, and
    -- items?
    -- The basic = is used for comparison of items.
    -- See also is_equal_map.
    require
        other_not_void: other /= Void
    ensure

```

```

    Result implies lower = other.lower and upper = other.upper;
    consistent: standard_is_equal(other) implies Result;
    symmetric: Result implies other.is_equal(Current)
is_equal_map (other: ARRAY[E]): BOOLEAN
    -- Do both collections have the same lower, upper, and
    -- items?
    -- Feature is_equal is used for comparison of items.
    -- See also is_equal.
    ensure
        Result implies lower = other.lower and upper = other.upper
all_default: BOOLEAN
    -- Do all items have their type's default value?
same_items (other: COLLECTION[E]): BOOLEAN
    -- Do both collections have the same items? The basic = is used
    -- for comparison of items and indices are not considered (for
    -- example this routine may yield true with Current indexed in
    -- range [1..2] and other indexed in range [2..3]).
    require
        other /= Void
    ensure
        Result implies count = other.count
occurrences (element: E): INTEGER
    -- Number of occurrences of element using equal for comparison.
    -- Also consider fast_occurrences to choose the most appropriate.
    ensure
        Result >= 0
fast_occurrences (element: E): INTEGER
    -- Number of occurrences of element using basic = for comparison.
    -- Also consider occurrences to choose the most appropriate.
    ensure
        Result >= 0
feature(s) -- Printing:
    fill_tagged_out_memory
        -- Append a viewable information in tagged_out_memory in
        -- order to affect the behavior of out, tagged_out, etc.
feature(s) -- Agents based features:
    do_all (action: ROUTINE[ANY,TUPLE[E]])
        -- Apply action to every item of Current.
    for_all (test: PREDICATE[ANY,TUPLE[E]]): BOOLEAN
        -- Do all items satisfy test?
    exists (test: PREDICATE[ANY,TUPLE[E]]): BOOLEAN
        -- Does at least one item satisfy test?
feature(s) -- Other features:
    get_new_iterator: ITERATOR[E]
    replace_all (old_value, new_value: E)
        -- Replace all occurrences of the element old_value by new_value
        -- using equal for comparison.
        -- See also fast_replace_all to choose the appropriate one.
    ensure
        count = old count;
        occurrences(old_value) = 0
    fast_replace_all (old_value, new_value: E)
        -- Replace all occurrences of the element old_value by new_value
        -- using operator = for comparison.
        -- See also replace_all to choose the appropriate one.
    ensure
        count = old count;

```

```

        fast_occurrences(old_value) = 0
move (lower_index, upper_index, distance: INTEGER)
-- Move range lower_index .. upper_index by distance
-- positions. Negative distance moves towards lower indices.
-- Free places get default values.
require
  lower_index <= upper_index;
  valid_index(lower_index);
  valid_index(lower_index + distance);
  valid_index(upper_index);
  valid_index(upper_index + distance)
ensure
  count = old count
slice (min, max: INTEGER): ARRAY[E]
-- New collection consisting of items at indexes in [min..max].
-- Result has the same dynamic type as Current.
-- The lower index of the Result is the same as lower.
require
  lower <= min;
  max <= upper;
  min <= max + 1
ensure
  same_type(Result);
  Result.count = max - min + 1;
  Result.lower = lower
reverse
-- Reverse the order of the elements.
ensure
  count = old count
feature(s)
capacity: INTEGER
-- Internal storage capacity in number of item.
subarray (min, max: INTEGER): ARRAY[E]
-- New collection consisting of items at indexes in [min .. max].
-- Result has the same dynamic type as Current.
-- See also slice.
require
  lower <= min;
  max <= upper;
  min <= max + 1
ensure
  Result.lower = min;
  same_type(Result);
  Result.count = max - min + 1;
  Result.lower = min or Result.lower = 0
feature(s) -- Interfacing with C:
to_external: POINTER
-- Gives C access into the internal storage of the ARRAY.
-- Result is pointing the element at index lower.
--
-- NOTE: do not free/realloc the Result. Resizing of the array
--       can makes this pointer invalid.
require
  not is_empty
ensure
  Result.is_not_null
feature(s) -- Modification:

```

```

resize (min_index, max_index: INTEGER)
  -- Resize to bounds min_index and max_index. Do not lose any
  -- item whose index is in both [lower .. upper] and
  -- [min_index .. max_index]. New positions if any are
  -- initialized with the appropriate default value.
  require
    min_index <= max_index + 1
  ensure
    lower = min_index;
    upper = max_index
reindex (new_lower: INTEGER)
  -- Change indexing to take in account the expected new_lower
  -- index. The upper index is translated accordingly.
  ensure
    lower = new_lower;
    count = old count
invariant
  valid_bounds: lower <= upper + 1;
  capacity >= upper - lower + 1;
  capacity > 0 implies storage.is_not_null;
end of ARRAY[E]

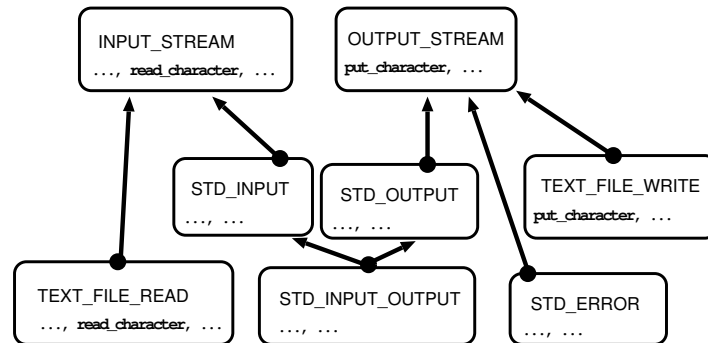
```

## 8 Entrées sorties simples

Les classes `STD_INPUT`, `STD_OUTPUT`, `STD_ERROR` et `STD_INPUT_OUTPUT` permettent d'effectuer des entrées sorties simples, sur les fichiers d'entrées/sorties standards (graphe d'héritage de la vue 65). Par défaut avec SmartEiffel, les variables `std_input`, `std_output`, `std_error` et `io` sont prédéfinies et référencent automatiquement les instances correspondant aux classes précédemment citées.

**Exercice 15** Ajouter la fonction `distance` dans la classe `POINT`. Cette fonction calcule la distance entre le point receveur et un deuxième point passé en argument.

Vue 65



## Entrées sorties

```

io.put_string("Entrez un nombre : ")
io.read_real
r := io.last_real
  
```

Vue 66

```

io.put_string("Entrez une seule lettre : ")
io.read_character
c := io.last_character
  
```

```

io.put_string("Entrez un mot : %N")
io.read_word
s := io.last_string
  
```



```

...
  from
    i := tab.lower
  until
    i > tab.upper
  loop
    io.put_string("Entrez une ligne : %N")
    io.read_string
    tab.put(io.last_string.twin,i)
--                                     *****
    i := i + 1
  end
...

```

cf. code Eiffel de `read_string` et `last_string`

## 9 Les assertions

Les mécanismes d'assertions du langage Eiffel permettent de documenter, et de faciliter la mise au point des programmes. Ces mécanismes permettent d'inclure dans le programme Eiffel une partie du travail fait au moment des spécifications (invariant, pré-condition, post-condition et invariant d'itération).

### 9.1 Invariant, pré- et post-condition

En théorie, si les pré-conditions d'une routine `r` d'une classe `C` sont satisfaites lorsque la routine est appelée, la classe garantit de délivrer un état où les post-conditions sont satisfaites. Les assertions constituent une sorte de *contrat* entre le client d'une classe et la classe elle-même, le fournisseur. On parle dans ce cas de *programmation par contrat*.

Les vues 69, 70, 71 et 72 donnent un modèle d'utilisation des assertions sur l'exemple classique de la classe `PILE_FIXE`. Cette classe modélise la notion de pile d'entiers de taille fixe.

Notons sur la vue 71 l'utilisation possible du mot clef `old` (uniquement possible dans une clause `ensure`). Sur la même vue, il faut remarquer que la deuxième assertion est incompréhensible par un client (utilisation de primitives non exportées). Il faut, autant que possible éviter d'utiliser autre chose que des primitives publiques (de consultation) dans les assertions. Et pire encore, il ne faudrait jamais utiliser de *modificateurs* dans une assertion : un programme «au point» **doit** aussi fonctionner sans le code des assertions.

**Exercice 16** Écrire la classe `PILE` qui modélise la notion de pile d'entiers non bornée.

Les invariants de classe expriment les propriétés globales d'une instance qui doivent être préservées par toute routine de la classe. Un invariant (vue 72) ne peut porter que sur l'état d'un objet (défini par les valeurs de ses attributs).

Les pré-conditions (**require**) et les post-conditions (**ensure**) sont prises en compte dans l'ordre de lecture du texte source Eiffel.

**Exercice 17** Écrire une nouvelle version de la classe POINT en :

- mémorisant les coordonnées polaires au lieu des coordonnées cartésiennes,
- ajoutant toutes les assertions qui conviennent,
- et en faisant attention de ne pas imposer de modifications aux clients existants

**Exercice 18** Quel est votre avis sur le fragment de code suivant ? erreur à la compilation ? erreur à l'exécution ? défaillance (temporaire) de l'auteur du programme ?

```

...
distance(p2 : POINT) is
  require
    Current /= Void
  do
  ...

```

## 9.2 Vérification ponctuelle

Il est également possible d'insérer des vérification ponctuelles à l'intérieur même d'une routine en utilisant une clause **check** ou une clause **debug**. Comme une clause **require**, la clause **check** contient une suite d'expressions booléennes à vérifier (vue 74).

La clause **debug** contient une suite d'instructions pouvant par exemple servir à placer des mouchards (vue 75). Bien entendu, il est important de bien faire attention aux effets de bord de façon à ne pas trop perturber le fonctionnement du programme. En effet, comme toutes les assertions, le mode de compilation conditionne la prise en compte (l'exécution) de ces instructions.

## 9.3 Assertion pour les itérations

La vue 76 donne le canevas syntaxique d'une itération et de ses assertions (clause **variant** et **invariant**). Les vues suivantes (78 et 79) donnent un exemple d'utilisation des assertions avec l'algorithme connu du **pgcd** (Plus Grand Commun Diviseur). L'invariant d'itération doit être respecté à chaque entrée dans le corps de l'itération (exactement comme dans le cas des preuves de programme). La clause **variant** concerne la preuve de terminaison de l'itération. C'est une expression entière, positive ou nulle qui doit décroître strictement après chaque passage dans l'itération.

## 9.4 Mise en œuvre

Sans toucher au texte source Eiffel d'une application (c'est un aspect essentiel), il est possible de décider quelles sont les assertions que l'on souhaite ou non effectivement vérifier lors de l'exécution. La commande de compilation **compile** permet de sélectionner les assertions que l'on souhaite vérifier (vue 77).

A l'exécution, en mode non optimisé, l'invariant est vérifié après chaque instanciation de la classe et après l'exécution de chaque routine de la classe.

## 9.5 Limites du mécanisme

Par rapport à l'approche purement théorique, la mise en œuvre des assertions en Eiffel possède ses propres limites. Dans le cas d'assertions récursives, afin de ne pas boucler indéfiniment, le système de gestion des assertions à l'exécution est libre de faire ou non certaines vérifications. Exemple : l'invariant du `pgcd` (vues 78 et 79).

Le code exécutable des assertions se déclenche à un instant bien précis alors que dans l'approche théorique, les assertions doivent être vérifiées à chaque instant. Ce faisant, il est possible d'écrire un programme Eiffel tel qu'on aboutisse à une situation aberrante : tous les mécanismes d'assertions sont en fonction, une assertion est violée et aucun message d'erreur ne s'affiche ! Un exemple (pas si facile que ça) est donné sur les vues 80, 81 et 82.

### Assertions Eiffel

*Valider, documenter et faciliter la mise au point d'une classe.*

*Contrat client/fournisseur.*

Vue 68 **invariant** *propriétés d'une instance*

**pré-conditions** require *contraintes sous lesquelles une routine fonctionne correctement*

**post-conditions** ensure *propriétés de l'état résultant de l'exécution d'une routine*

Vue 69

```

class PILE_FIXE -- Pile d'entiers de taille limitée.
creation {ANY}
  make
feature {NONE} -- Attributs privées :
  table : ARRAY[INTEGER]
  nb_elements : INTEGER
feature {ANY} -- Primitives de consultation :
  vide : BOOLEAN is
    do
      Result := (nb_elements = 0)
    end -- vide
  pleine : BOOLEAN is
    do
      Result := (nb_elements = table.count)
    end -- pleine

```

Vue 70

```

  sommet : INTEGER is
    require
      not Current.vide
    do
      Result := table.item(nb_elements)
    end -- sommet
feature {ANY} -- Modifications :
  empiler(val : INTEGER) is
    require
      not Current.pleine
    do
      nb_elements := nb_elements + 1
      table.put(val, nb_elements)
    ensure
      not Current.vide
    end -- empiler

```

Vue 71

```
depiler is
  require
    not vide
  do
    nb_elements := nb_elements - 1
  ensure
    nb_elements + 1 = old nb_elements
    --
    nb_elements /= table.count
  end -- depiler
```

Vue 72

```
make(tailleMaxi : INTEGER) is
  require
    tailleMaxi > 1
  do
    create table.make(1, tailleMaxi)
  ensure
    Current.vide
  end -- make
invariant
  nb_elements >= 0
  nb_elements <= table.count
end -- class pile_fixe
```

## Interface de la classe PILE\_FIXE

Vue 73

```

class PILE_FIXE -- Pile d'entiers de taille limitée.
feature {ANY} -- Primitives de consultation :
  vide : BOOLEAN is
  pleine : BOOLEAN is
  sommet : INTEGER is
    require
      not Current.vide
feature {ANY} -- Modifications :
  empiler(val : INTEGER) is
    require
      not Current.pleine
    ensure
      not Current.vide
  ...

```

## check : vérifications ponctuelles

*Des assertions au cœur d'une routine*

Vue 74

```

local
  p1, p2 : POINT
do
  create p1.make(0,0)
  p2 := p1.twin
  p1.translater(+2,+4)
  p1.translater(-2,-4)
  check
    p1.is_equal(p2)
    p1.distance(p2) = 0
end

```

**debug : placer un mouchard**

Vue 75

```
local
  i: INTEGER
do
  ...
  debug
    if i < 0 then
      io.put_string("Attention i < 0")
    end
  end
  ...
```

**Assertions dans une itération**

Vue 76

```
...
  from
  invariant
    expressions-booléennes
  variant
    expression-entière
  until
    ...
  loop
    ...
  end
  ...
```

## Options de compilation

compile  
 [-debug\_check|-all\_check|-loop\_check|  
 -invariant\_check|-ensure\_check|  
 -require\_check|-no\_check|-boost]

Vue 77

[-cc] [-no\_strip] [-c\_code]  
 [-o output\_name]  
 <ROOT\_CLASS> start\_procedure  
 [\*c] [\*o] [-l\*]

<http://SmartEiffel.loria.fr>

gcd(other: INTEGER): INTEGER is  
 -- Great Common Divisor of 'Current' and 'other'.  
 require  
 Current > 0; other > 0  
 local  
 the\_other: INTEGER  
 do  
 from  
 Result := Current  
 the\_other := other  
 invariant  
 Result > 0  
 the\_other > 0  
 Result.gcd(the\_other) = Current.gcd(other)

Vue 78



Vue 79

```

variant
  Result.max(the_other)
until
  Result = the_other
loop
  if Result > the_other then
    Result := Result - the_other
  else
    the_other := the_other - Result
  end
end
ensure
  Result = other.gcd(Current)
end

```

## Limite des assertions

Vue 80

```

class A
feature {ANY}
  aller : B

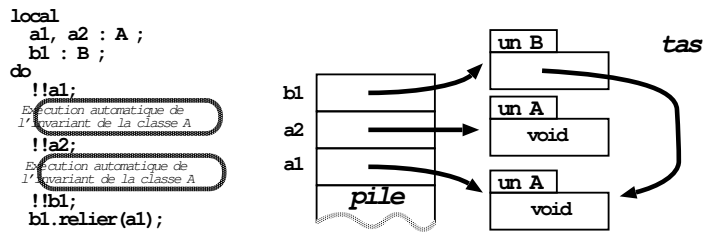
  relier (b1 : B) is
  do
    aller := b1
  end -- relier

invariant
  aller = Void or else
  (aller.retour = Current)
end -- class A

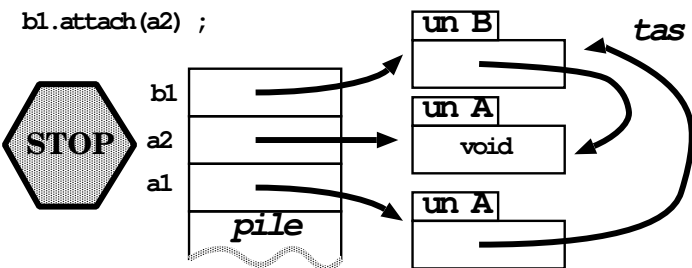
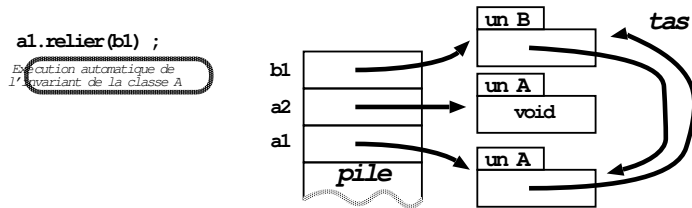
class B
feature {ANY}
  retour : A

  relier (a1 : A) is
  do
    retour := a1
  end -- relier
end -- class B

```



Vue 81



Vue 82

Invariant non vérifié et non détecté pour l'instance référencée par a1.

## 10 Les outils

Les outils disponibles actuellement sont les commandes `compile`, `compile_to_c`, `finder`, `short`, `class_check` et `pretty` (vue 84). Si vous disposez de Java sur votre ordinateur personnel, vous pouvez aussi utiliser les commandes `compile_to_jvm` ainsi que `print_jvm_class`.

### 10.1 Le mode emacs

Il existe plusieurs modes `emacs` disponibles soit dans la distribution de SmartEiffel (cf. `SmartEiffel/misc/eiffel.el`), soit avec `xemacs` sous Linux, soit télé-chargeables sur internet.

Voici la liste des commandes spécifiques du mode Eiffel sous `emacs` (`ESC ? b CTRL-x o`) :

Local Bindings:

key	binding
---	-----
RET	eiffel-return
TAB	eiffel-indent-line
C-c C-e	class-compile
C-c t	eiffel-line-type
C-c e	eiffel-elsif
C-c w	eiffel-when
C-c n	eiffel-inspect
C-c l	eiffel-loop
C-c i	eiffel-if
C-c a	eiffel-attribute
C-c p	eiffel-procedure
C-c f	eiffel-function
C-c c	eiffel-class
ESC ;	eiffel-comment

### 10.2 Les commandes du débogueur

Pour exécuter un programme avec `sedb`, le débogueur de SmartEiffel, il suffit de recompiler son programme en ajoutant l'option `-sedb` puis de relancer l'exécution comme d'habitude. L'exécution sous `sedb` commence toujours avec un écran donnant la liste des commandes disponibles (vue 85). Le débogueur `sedb` attend votre première commande. Ces commandes permettent de suivre l'exécution au pas par pas d'un programme en visualisant la pile ainsi que le texte source du programme Eiffel lors de l'exécution.

En cas d'erreur, le débogueur s'arrête automatiquement dans la contexte précis de l'erreur (violation d'une assertion par exemple). Il est possible de définir des points d'arrêt sur la base de plusieurs conditions et aussi de placer explicitement des points d'arrêt directement dans le code source Eiffel. Il est également possible d'interrompre l'exécution d'un programme qui *boucle* ou encore d'effectuer des mesures concernant le nombre d'appels d'une routine particulière. Le débogueur fait habituellement l'objet d'une présentation interactive en cours, sur écran projeté, à ne pas manquer. . .

### short : donne l'interface

```
class interface TRIANGLE
creation {ANY}
  make
Vue 83 feature {ANY}
  p1, p2, p3: POINT
  make(initP1, initP2, initP3: POINT)
  translater (dx, dy: REAL)
  ...
end interface -- class TRIANGLE
```

### Les outils

```
emacs / xemacs / mode Eiffel
compile -sedb
Vue 84 compile_to_c
  pretty
  short
  class_check
  finder
```

SmartEiffel debugger help (list of commands):

```

?      : display this help message
s      : single step, stepping into routine calls
n      : single step, without stepping into routine calls
f      : continue execution until the current routine finishes
.      : display the current execution point
u      : display the caller point (go up in the stack)
d      : display the callee point (go down in the stack)
S      : display the entire stack (all frames)
b      : set a new breakpoint at the current execution point
B      : display the entire breakpoint list
-[#]   : delete a breakpoint
c      : continue execution until the next breakpoint
G      : run the garbage collector now
T      : switch to the "trace.se" file mode
q      : quit the debugger
H      : display more help on using the debugger
Enter: repeat the previous debugger command

```

Vue 85

## 11 Bien programmer en Eiffel

Comment programmer en respectant autant que possible l'esprit dans lequel a été défini le langage Eiffel ? Je n'ai malheureusement pas la réponse systématique à cette question, mais j'essaye de donner dans la suite une liste de conseils à respecter autant que possible (vue 86).

Avant de rendre un T.P., pensez à vérifier si votre programme respecte bien ces règles. Si ce n'est pas le cas (c'est possible), justifiez vous !

Le premier principe à respecter consiste à n'écrire **que des méthodes** (vue 91) . Comment vérifier qu'une routine est véritablement une méthode ? C'est simple, il suffit de regarder si le corps de la routine utilise effectivement **Current** ou bien un (ou des) attribut(s) de **Current**. D'une manière générale, si un algorithme concerne essentiellement une instance de la classe POINT, cet algorithme doit être dans la classe POINT et l'algorithme doit s'appliquer au receveur (**Current**).

Le langage Eiffel incite à agir directement sur le receveur (vue 94). En général, on a intérêt à modifier directement le receveur par une procédure.

**Exercice 19** Considérons un programme client de la classe POINT qui référence un point existant par l'intermédiaire de la variable `p1`. Ce client souhaite modifier le point `p1` par translation. Si on utilise la procédure `translater` de la vue 19 il faut exécuter l'instruction :

```
p1.translater(2,2)
```

Écrire une (ou des) instructions aboutissant au même résultat en utilisant la procédure `translater2` de la vue 94. Même question avec la procédure `translater3`. Comparez les trois solutions.

**Exercice 20** Le client de la classe `POINT` souhaite maintenant (cf. exercice précédent) conserver son point initial référencé par `p1` et obtenir un nouveau point. En utilisant `translater2`, écrire l'équivalent de :

```
p2 := p1.twin
p2.translater(2,2)
```

Même question pour `translater3`. Comparer les trois solutions.

**Exercice 21** Reprendre les deux exercices précédents avec la très étrange (et très mauvaise) solution suivante :

```
translater4 (dx, dy : REAL) : POINT is
do
  x := x + dx
  y := y + dy
  Result := Current
end -- translater
```

Il est essentiel de cacher autant que possible ce qui ne concerne pas véritablement les clients d'une classe (vues 87 et 88). Dans la mesure du possible, il est souhaitable de localiser et de minimiser les endroits où l'on travaille directement sur la partie privée d'une classe (vues 89 et 90).

## les 6 (premiers) commandements

cacher l'implantation

définir de vraies méthodes

Vue 86 choisir des sélecteurs uniformes

modifier le receveur

singer le monde réel

mettre en facteur

## cachez l'implantation

*Le client ne sait pas si un point est mémorisé par ses coordonnées cartésiennes ou par ses coordonnées polaires*

```
Vue 87      pt : POINT
            ...
            val := pt.x
            val := pt.y
            val := pt.rho
            val := pt.theta
```

*Ne pas exporter ce qui ne concerne pas le client :*

```
class TRIANGLE
...
feature {NONE}
  tab : ARRAY[POINT]
...
```

Vue 88

*Le fournisseur utilise en priorité son interface :*

**Mauvais :**

```
translater (dx,dy : REAL) is
  do
    tab.item(1).translater(dx,dy)
    tab.item(2).translater(dx,dy)
    tab.item(3).translater(dx,dy)
  end -- translater
```

*Minimiser et localiser les endroits où l'on travaille directement sur la partie privée*

```
class TRIANGLE
...
feature {ANY}
  p1 : POINT is
    do Result := tab.item(1)
    end
  p2 : POINT is
    do Result := tab.item(2)
    end
  p3 : POINT is
    do Result := tab.item(3)
    end
```

Vue 89



*Utiliser l'interface dans les assertions de l'interface!*

Vue 90

```

class TRIANGLE
...
invariant
    p1 /= Void
    p2 /= Void
    p3 /= Void
end

```

### définir de vraies méthodes

*Une méthode doit concerner le receveur : Current ou ses attributs.*

**Très mauvais exemple :**

Vue 91

```

class POINT
...
    translater_triangle(dx, dy : REAL; t : triangle) is
        do
            t.p1.translater(dx,dy)
            t.p2.translater(dx,dy)
            t.p3.translater(dx,dy)
        end

```

*Une routine qui manipule un triangle doit se trouver dans la classe TRIANGLE et agir sur le receveur*

### choisir des sélecteurs uniformes

*Un sélecteur ne doit pas mentionner le nom de la classe où se trouve la routine. Mauvais : `translater_triangle`,*

Vue 92 *`translater_point`, etc.*

*D'une classe à l'autre, il faut retrouver le même sélecteur : `translater` dans la classe POINT, `translater` dans la classe TRIANGLE, `translater` dans la classe FIGURE, etc.*

Vue 93

```
class POINT
creation {ANY}
  make, lire, test
...
feature {ANY}
  make(vx, vy : REAL) is
    do
      x := vx
      y := vy
    end -- make

test is -- Petit test de la classe POINT :
...
lire (file: TEXT_FILE_READ) is
...
```

## modifier le receveur

*Travailler par modification du receveur sans instancier un nouvel objet.*

**Mauvais :**

```
class POINT
...
    translater2 (dx, dy : REAL) : POINT is
    do
        create Result.make(x + dx, y + dy)
    end -- translater

    translater3 (dx, dy : REAL; p2 : POINT) is
    do
        p2.make(x + dx, y + dy)
    end -- translater
```

Vue 94

## singer le monde réel

*Autant d'instances lors de l'exécution qu'il y a d'objets dans le monde réel*

Vue 95 *Autant d'instances de la classe VOITURE dans l'instance de la classe PARKING que de voitures sur le parking*

*Exception : pour économiser un trop grand nombre d'instances (Flyweight Design Pattern)*

### mettre en facteur

Vue 96 *Eviter les redondances dans les données ainsi que dans le texte source Eiffel*

*Introduire des **routines** auxiliaires.*

*Utilisez des constantes*

### vérifier aussi que :

*Les fonctions sont sans effet de bord*

*L'interface d'une classe est facile à comprendre.*

Vue 97 *Une classe est facile à utiliser. Par exemple, il faut éviter d'exporter une routine d'initialisation*

*Ne pas gaspiller la mémoire : évitez les instanciations inutiles.*

*Une classe ne porte pas le nom d'un traitement : classe ANALYSE, classe TRANSLATER, etc.*

Penser également à présenter correctement le texte source Eiffel en respectant l'indentation standard (celle de ce polycopié ou celle préconisée dans ETL). Prendre l'habitude de classer les primitives selon qu'il s'agit d'observateurs ou de modificateurs (vue 98).

## Observateur / Modificateur

```

class interface PILE
feature {ANY} -- Observateurs :
  vide : BOOLEAN
  pleine : BOOLEAN is
  sommet : INTEGER is
    require
      not vide
feature {ANY} -- Modificateurs :
  empiler(val : INTEGER) is
    require
      not pleine
    ensure
      not vide

```

Vue 98

## 12 Routines à exécution unique

Pour définir une routine à exécution unique, il suffit de remplacer le mot clé **do** par le mot clé **once** dans la définition de la routine. La routine obtenue ne s'exécute, au plus, qu'une seule fois (vue 99). S'il s'agit d'une fonction, le résultat calculé lors de la première exécution est mémorisé dans la partie données statiques de la mémoire (vue 27). Ce résultat mémorisé est retourné lors des appels suivants.

**Exercice 22** Pour les amateurs de langage machine uniquement. Imaginer une traduction systématique en code machine Motorola MC68000 pour les routines **once**. Indications : un booléen peut être associé à chaque routine **once** dans la zone des données allouées statiquement. L'inspection du code C généré par SmartEiffel peut servir aussi.

**Exercice 23** Reprendre l'exercice précédent et optimiser le code généré en utilisant un pointeur de fonction ainsi que du code automodifiant. Indication : il s'agit de supprimer des conditionnelles dans le code généré.

## Routines ONCE

*Définition de routines à exécution unique*

```

class FENETRE
feature {NONE}
  ...
  initialisation is
    once
      -- Instructions d'initialisation à
      -- n'exécuter qu'une seule fois.
      ...
    end
  ...

```

Vue 99

```

feature {ANY}
  ouvrir_cadre (...) is
    do
      initialisation
      ...
    end
  ouvrir_icone (...) is
    do
      initialisation
      ...
    end
end -- class FENETRE

```

Vue 100

```

class TEST_ONCE
creation {ANY}
  make
feature {ANY}
  pi : REAL is
    once
      Result := 3.14159
    end

  origine : POINT is
    once
      create Result.make(0,0)
    end

```

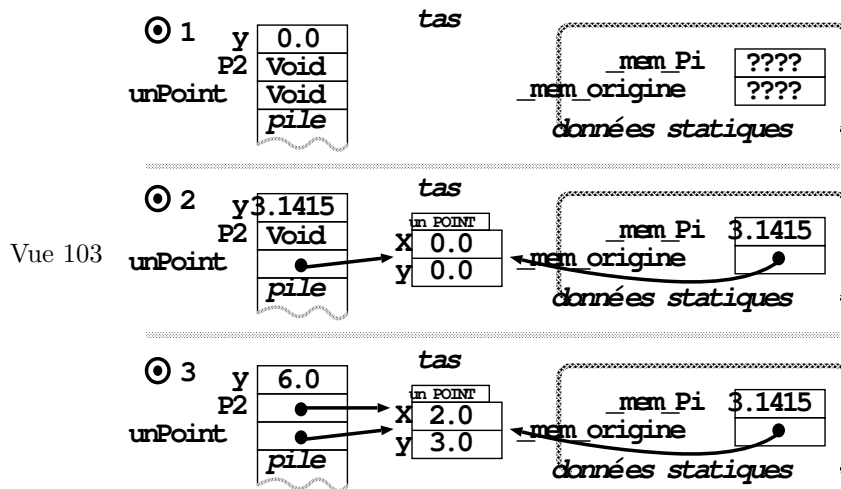
Vue 101

```

make is
  local
    unPoint, p2 : POINT
    y : REAL
  do -- ⊙ 1
    unPoint := Current.origine
    y := Current.pi -- ⊙ 2
    y := 6.
    unPoint.translater(2,3)
    p2 := Current.origine -- ⊙ 3
  end -- make
end -- class TEST_ONCE

```

Vue 102



Vue 104

```

class QUI_NUMEROTE_SES_INSTANCES
creation make
feature {ANY}
    numero: INTEGER
feature {NONE}
    counter: COUNTER is
        once
            create Result
        end
    make is
        do
            numero := counter.value
            counter.increment
        end
end
end
    
```

cf. SmartEiffel/lib\_show/random/demol



## 13 Divers

### 13.1 Priorité des opérateurs

#### Priorité des opérateurs

Priorité	
12	. agent
11	old not +(unaire) -(unaire) <i>autres_opérateurs_unaires</i>
10	<i>opérateurs_binaires_non_standards</i>
9	^(puissance)
8	* // (division entière) \\ (reste)
7	+(binaire) -(binaire)

Vue 105

Vue 106

6	=	/(non égal)	<	>	<=	>=
5	and	and then				
4	or	xor	or else			
3	implies					
2	[]	(tuple explicite)				
1	;	(dans les assertions)				

**Exemples :**

$$\begin{array}{c}
 x := y^z * t + v \\
 \Downarrow \\
 x := ((y^z) * t) + v
 \end{array}$$

$$\begin{array}{c}
 x := a.b + c \\
 \Downarrow \\
 x := ((a.b) + c)
 \end{array}$$

Vue 107

$$\begin{array}{c}
 \text{val} := \text{tab} @ i . c \\
 \Downarrow \\
 \text{val} := (\text{tab} @ (i . c))
 \end{array}$$

$$\begin{array}{c}
 a.b.c(d).e.f \\
 \Downarrow \\
 (((a.b).c(d)).e).f
 \end{array}$$

## 13.2 Attributs constants

Pour tous les types de base (vue 25), ainsi que la classe `STRING`, la syntaxe montrée sur la vue 108 permet la définition d'attributs constants.

Il faut être conscient du fait qu'une constante de type `STRING` est l'équivalent d'une routine `once` (vue 109 et 110). Il n'y a donc qu'une et une seule instance en mémoire pour une chaîne `STRING` écrite de manière explicite.

### attributs constants

```
Vue 108      pi : REAL is 3.14159265358979323846
             Stdout : INTEGER is 0
             Stderr : INTEGER is 1
             mois_09 : STRING is "Septembre"
             epsilon : DOUBLE is 0.000000000000002
             vert : CHARACTER is 'v'
             new_line : CHARACTER is '%N'
             perdu : BOOLEAN is False
             texte : STRING is "Ceci est un texte %
                           %qui est long."
             saute_3_lignes : STRING is "%N%N%N"
             quote_simple : CHARACTER is '%''
```

**Attention !**

```
mois_09 : STRING is "Septembre"
```

⇕

Vue 109

```
mois_09 : STRING is
  once
    Result := "Septembre"
  end
```

```
-----
mois_09.put('Z',1)
```

```
...
message := "Essai 1%N"
```

⇕

```
str000257 : STRING is "Essai 1%N"
```

Vue 110

```
...
message := str000257
```

```
-----
message := "Essai 1%N"
io.putstring(message)
message.put('2',7)
io.putstring(message)
```



### create en tant qu'expression

*La nouvelle notation permet de créer de nouveaux objets, en évitant l'utilisation de variables supplémentaires (expression/instruction) :*

```
-- Appel de la procédure foo avec un nouveau POINT:
foo ( create {POINT}.make(1,1) )
```

Vue 112

```
-- Appel de la procédure bar avec un nouveau TRIANGLE:
bar ( create {TRIANGLE}.make (
                                create {POINT}.make(1,1),
                                create {POINT}.make(2,2),
                                create {POINT}.make(3,3)
                                )
```

## 14 Exercices/anciens sujets d'examen

**Exercice 25** En utilisant la primitive `distance` qui donne la distance entre deux points, donner une nouvelle version de la primitive `d2ao` de la vue 19. Quels sont les avantages et les inconvénients de cette solution ? Comment faire pour éviter d'instancier un point lors de chaque appel ?

**Exercice 26** Comment la classe `ARRAY` est-elle vraiment implantée ? Indication (et réponse) : consulter le texte source `array.e`.

**Exercice 27** Écrire complètement en Eiffel, sans utiliser les tableaux du langage C, une autre implantation de la classe `ARRAY`<sup>1</sup>. Les possibilités offertes par l'implantation actuelle doivent être conservées (comme par exemple, la possibilité d'étendre un tableau).

**Exercice 28** Écrire une nouvelle version de la classe `STRING` en utilisant la classe `ARRAY`.

**Exercice 29** Exécutez le programme suivant sur la machine :

```
...
from
  i := 1
```

---

<sup>1</sup>Exercice difficile et à garder pour le deuxième semestre

```

until
  x > 3
loop
  s := once "hop"
  s.append("hop")
  io.put_string(s)
  i := i + 1
end
...

```

Est-ce le résultat attendu ?

Pour bien comprendre ce qui se passe, refaites l'essai en enlevant le mot clef **once** qui est devant la chaîne "hop".

**Exercice 30** Combien de chaînes de caractères de la classe `STRING` sont instanciées durant l'exécution du fragment de programme suivant :

```

io.read_string
s1 := io.last_string
io.read_string
s2 := io.last_string

```

**Exercice 31** Écrire une nouvelle version de la classe `TRIANGLE` telle que :

- les trois points constituant le triangle soient mémorisés grâce à une instance de la classe `ARRAY`,
- l'interface de la nouvelle version de la classe `TRIANGLE` soit identique à la précédente.

**Exercice 32** Soit le fragment de programme Eiffel suivant :

```

...
local
  pt1, pt2, pt3, pt4 : POINT
  t1, t2, t3 : TRIANGLE
do
  create pt1.make(1,1)
  create pt2.make(2,2)
  create pt3.make(3,3)
  create t1.make(pt1,pt2,pt3)
  t2 := t1
  t1 := t3
  create pt1.make(1,1)
  pt1 := pt4
  create pt1.make(1,1)
  pt1 := pt4
  pt2 := pt4
  ⊙ ← Point d'arrêt ici.

```

Répondez aux questions suivantes en considérant que l'exécution est momentanément suspendue au point d'arrêt indiqué à la fin du fragment de programme.

- a) Donnez pour les classes `POINT` d'une part et `TRIANGLE` d'autre part, le nombre total d'instanciations effectuées.
- b) Quelles sont les variables locales du fragment de programme qui ne référencent aucune instance ?
- c) Combien y-a-t-il d'instances de la classe `POINT` potentiellement accessibles ? De quelle façon ?

d) Si l'on considère qu'une instance de la classe POINT occupe deux mots mémoire et qu'une instance de la classe TRIANGLE en occupe trois, quel est le nombre minimum de mots mémoire nécessaire à l'exécution de ce fragment de programme ? On suppose bien entendu que le ramasse-miettes est en parfait état de fonctionnement.

**Exercice 33** On désire disposer d'un moyen rapide d'accès et de calcul des différents termes de la fonction `factoriel` en évitant les (re)calculs inutiles. Voici l'interface de la classe Eiffel chargée de cette tâche :

```
class interface FACTORIEL
feature {ANY}

    i_th (n: INTEGER): INTEGER
        require
            n >= 0

end interface -- class FACTORIEL
```

L'unique fonction exportée, `i_th` permet à chaque appel d'obtenir la valeur correspondant à `factoriel n`. Par exemple :

```
x : INTEGER
fact : FACTORIEL
do
    create fact
    x := fact.i_th(2)
    -- actuellement, x contient 2
    x := fact.i_th(3)
    -- actuellement, x contient 6
    x := fact.i_th(2)
    -- actuellement, x contient 2
```

Écrire le texte Eiffel de la classe `FACTORIEL` correspondante en respectant scrupuleusement les contraintes suivantes :

- un même terme de la fonction `factoriel` ne doit jamais être calculé deux fois, autrement dit, une fois un résultat calculé il est automatiquement mémorisé,
- aucun terme de la fonction `factoriel` ne doit être calculé avant que cela ne soit véritablement nécessaire ;
- même s'il y a plusieurs instances de la classe `FACTORIEL`, la mémoire des résultats déjà calculés doit être unique.

**Exercice 34** Soit la classe `DOUBLET` :

```
class DOUBLET
creation {ANY}
    make
feature {ANY}
    premier: INTEGER
    reste: DOUBLET
    make(p: INTEGER; r: DOUBLET) is
        do
            premier := p
            reste := r
        end -- make
```



```

dupliquer : DOUBLET is
  do
    create Result.make(premier, reste)
  end -- dupliquer
change_reste (nr: DOUBLET) is
  do
    reste := nr
  end -- change_reste
end -- class doublet

```

Dessinez l'état de la mémoire lors de l'exécution du fragment de programme suivant :

```

...
local
  liste_a, liste_b, liste_c : DOUBLET
  duvoid, temp, temp2 : DOUBLET
do
  create temp.make(3, duvoid)
  create temp2.make(2, temp)
  create liste_a.make(1, temp2)

  temp := liste_a.reste.dupliquer
  create liste_b.make(1, temp)

  create temp.make(2, liste_a.reste.reste.dupliquer)
  create liste_c.make(1,temp)
  temp.reste.change_reste(liste_c)
  temp := Void
  temp2 := duvoid
  ⊙ ← ÉTAT DE LA MÉMOIRE ICI.

```

**Exercice 35** On considère à nouveau la classe DOUBLET utilisée précédemment. Complétez la définition de la fonction `question1` dont l'interface est définie comme suit :

```

question1: DOUBLET is
  local
    ...
  do
    ...
    ...
  ensure
    Result.premier = 1
    Result.reste.premier = 2
    Result.reste.reste.premier = 3
    Result.reste.reste.reste = Void
  end

```

**Exercice 36** Complétez la définition de la fonction `question2` dont l'interface est définie comme suit :

```

question2: DOUBLET is
  local
    ...
  do
    ...

```

```

...
ensure
  Result.premier = 1
  Result.reste.premier = 2
  Result.reste.reste.premier = 3
  Result.reste.reste.reste = Result
end

```

Indication : cette fonction construit et retourne une liste circulaire à trois éléments (1,2,3,1,2,3,1,2,3,...).

**Exercice 37** Dessinez (en respectant les conventions vues en cours) l'instance retournée par la fonction `question3` suivante :

```

question3: DOUBLET is
  do
    Result := question2
    Result.reste.reste.change_reste(Void)
  end

```

N'oubliez pas d'indiquer schématiquement (par une flèche) la valeur du résultat de la fonction `question3` elle-même.

**Exercice 38** Dessinez (en respectant les conventions vues en cours) l'instance retournée par la fonction `question4` suivante :

```

question4: DOUBLET is
  do
    Result := question2.dupliquer
  end

```

De même, n'oubliez pas d'indiquer schématiquement la valeur du résultat de la fonction `question4` elle-même.

**Exercice 39** Soit la classe A :

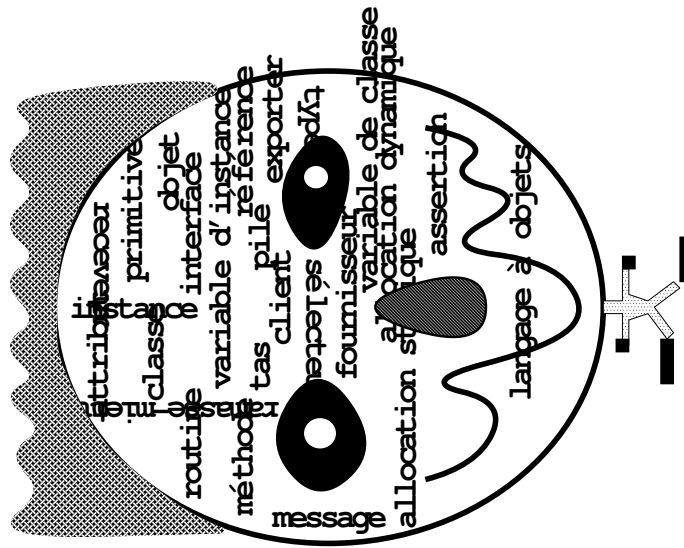
```

class A
  feature {ANY}
    lien : A
    attacher (unA : A) is
      do
        lien := unA
      end -- attacher
    allonger(combien : INTEGER) is
      local
        unA : A
        i : INTEGER
      do
        from
          i := combien
        until
          i = 0
        loop
          i := i - 1
          create unA

```



Vue 113



## 15 Généricité non contrainte

La généricité est le fait de pouvoir définir une classe dont la définition est paramétrée par une ou plusieurs autres classes. On parle de généricité non contrainte dans le cas où l'on ne fait aucune supposition particulière sur les opérations disponibles dans la (ou les) classe(s) paramètre(s). Les vues 114, 115, 116 et 117 présentent l'exemple de la classe PILE[T] qui permet la manipulation de piles d'entiers, de piles de caractères, de piles de piles, etc.

Dans le cas d'une pile, il n'est pas nécessaire de supposer quoi que ce soit sur le type des éléments à empiler. Aucune supposition particulière n'est faite concernant le type T. C'est donc un exemple de généricité non contrainte.

## GENERICITE non contrainte

```

class PILE[T]
creation {ANY}
  make
  -- Variables d'instances :
feature {NONE}
  table : ARRAY[T]
feature {ANY}
  nb_elements : INTEGER
feature {ANY} -- Constructeur :
  make is
  do
    create table.make(1,2)
    nb_elements := 0

    ensure
      vide
    end -- make
feature {ANY} -- Consultations :
  vide : BOOLEAN is
  do
    Result := (nb_elements = table.lower - 1)
  end -- vide
  sommet : T is
  require
    not vide
  do
    Result := table.item(nb_elements)
  end -- sommet

```

Vue 114

Vue 115

Vue 116

```
feature {ANY} -- Modifications :
  empiler (x : T) is
    do
      nb_elements := nb_elements + 1
      table.force(x,nb_elements)
    ensure
      not vide
      sommet = x
      nb_elements = (old nb_elements) + 1
    end -- empiler
```

Vue 117

```
  depiler is
    require
      not vide
    do
      nb_elements := nb_elements - 1
    ensure
      nb_elements = (old nb_elements) - 1
    end -- depiler
  invariant
    nb_elements >= 0
    table.upper >= nb_elements
end -- class PILE
```

Une classe générique n'est pas utilisable directement mais uniquement par *dérivation*<sup>1</sup>, en fixant une valeur aux types paramètres.

Dans l'exemple de la classe PILE, T est un paramètre générique formel. Le paramètre générique effectif est donné lors de la déclaration de variables permettant la manipulation de piles (vue 118). Le type PILE[CHARACTER] est dit «génériquement dérivé» du type PILE[T] en utilisant CHARACTER comme paramètre générique effectif.

Le type PILE[CHARACTER] n'est pas conforme avec le type PILE[POINT]. Il est donc interdit d'affecter entre elles les variables `pileDeCaracteres` et `pileDePoints`.

## Dérivation d'un type générique

```

local
  pileDeCaracteres : PILE[CHARACTER]
  pileDePoints : PILE[POINT]
  pileDePileDePoints : PILE[PILE[POINT]]
do
  create pileDeCaracteres.make
  create pileDePoints.make
  create pileDePileDePoints.make
  pileDeCaracteres.empiler('a')
  if pileDePoints.vide then
    pileDePoints := pileDePileDePoints.sommet

```

Vue 118

La classe ARRAY prédéfinie en Eiffel est également une classe générique (vue 119).

La généricité représente un progrès considérable en évitant la duplication de textes sources. Tous les langages de classes ne disposent pas forcément de la généricité (vue 119) qui, comme nous le verrons plus tard, peut être plus ou moins «simulée» avec l'héritage.

<sup>1</sup>Dans la littérature, on utilise également le terme «instanciation d'un type générique» qui présente l'inconvénient de prêter confusion avec l'instanciation d'une classe.

```

local
  tab : ARRAY[CHARACTER]
  ...

```

Vue 119

	<i>genericité</i>	<i>typage statique</i>	<i>liaison dynamique</i>
Eiffel	oui	oui	oui
ADA95	oui	oui	depuis 1995
C++	template	oui	Pas toujours
Smalltalk	NON	NON	OUI
Java	non	oui	oui

**Exercice 41** La méthode `depiler` de la classe `PILE[T]` (vue 117) ne fait que décrémenter l'indice de la table qui continue donc à référencer l'objet dépilé ! Expliquez ce qui se passe lorsque le ramasse-miettes se déclenche ? Corriger cette méthode afin que le ramasse-miettes puisse faire correctement son travail s'il y a lieu.

**Exercice 42** Écrire une autre implantation la classe `PILE` en utilisant une liste chaînée (de la classe `LIST` par exemple).

**Exercice 43** En utilisant un traitement de textes ou en utilisant la commande `sed` sous UNIX, comment pourrait-on ajouter au langage C un mécanisme comparable (généricité non contrainte). Donnez les grands principes.

**Exercice 44** Écrire une commande UNIX capable de dresser la liste des classes génériques de la bibliothèque Eiffel.

## 16 Héritage



## HÉRITAGE

Réutilisation et mise en facteur de points communs entre différentes classes.

**classification – spécialisation**

Vue 120

**abstraction – factorisation d’algorithmes**

**implantation**

### Mécanisme

Recopie automatique de texte source

### 16.1 Spécialiser par enrichissement

On suppose qu’une première application de gestion d’un stock d’articles existe et fonctionne. Une classe de cette application, la classe `ARTICLE` est donnée sur les vues 121, 122 et 123.

```

class ARTICLE
creation {ANY}
  make
feature {ANY} -- Variables d'instances :
  designation : STRING
  prix_ht : REAL
  quantite : INTEGER
Vue 121 feature {ANY} -- Constructeur :
  make (nom : STRING prix : REAL; nombre : INTEGER) is
    require
      prix > 0; nombre >= 0
    do
      designation := nom
      prix_ht := prix
      quantite := nombre

    ensure
      designation = nom; prix_ht = prix
      quantite = nombre
    end -- make
feature {ANY} -- Consultations :
  prix_ttc : REAL is
    do
      Result := prix_ht * 1.186
    end -- prix_ttc
Vue 122 feature {ANY} -- Modifications :
  retirer (nombre : INTEGER) is
    require
      nombre > 0; nombre <= quantite
    do
      quantite := quantite - nombre

```

Vue 123

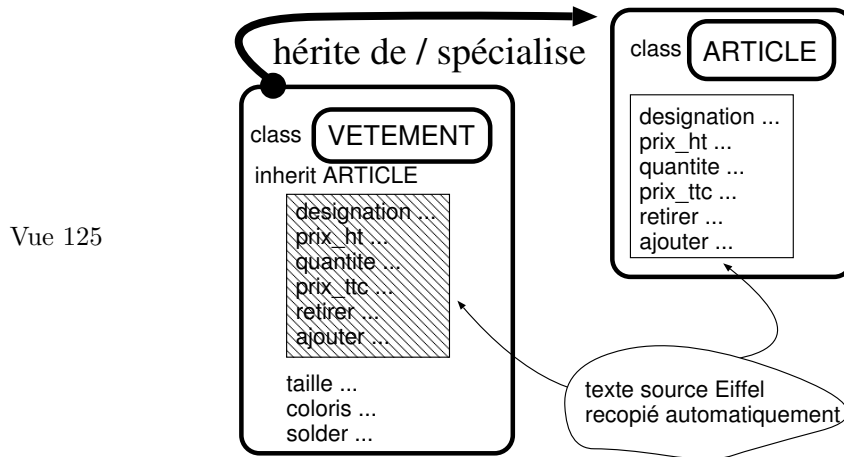
```
        ensure
            quantite = (old quantite) - nombre
        end -- retirer
    ajouter (nombre : INTEGER) is
        require
            nombre > 0
        do
            quantite := quantite + nombre
        ensure
            quantite = (old quantite) + nombre
        end -- ajouter
    invariant
        quantite >= 0; prix_ht >= 0
    end -- class ARTICLE
```

On suppose maintenant qu'il nous faut développer une deuxième application (pour un autre client par exemple). Cette deuxième application est similaire à la précédente car il s'agit de gérer un stock de vêtements. La notion de vêtement n'est qu'un cas particulier de la notion d'article plus générale et on souhaite que toutes les primitives de la classe ARTICLE (*designation*, *prix\_ht*, *quantite*, *ajouter* et *retirer*) soient reprises dans la classe VETEMENT. Un VETEMENT doit en outre disposer de primitives spécifiques (*taille*, *coloris*, *solder*). Dans ce cas, on dit que la classe VETEMENT est une **spécialisation** de la classe ARTICLE. L'héritage est bien adapté pour réaliser une telle relation de spécialisation (cf. vue 124).

```
class VETEMENT
inherit
  ARTICLE
creation {ANY}
  make
feature {ANY}
  taille : INTEGER
  coloris : STRING
  solder (remise : REAL) is
    require
      0 < remise; remise < 100
    do
      prix_ht := prix_ht * (1 - (remise / 100))
    ensure
      prix_ht = (old prix_ht) * (1 - (remise / 100))
    end -- solder
end -- class VETEMENT
```

Vue 124

La classe VETEMENT hérite de la classe ARTICLE. On dit dans ce cas que la classe ARTICLE est la **superclasse** de la classe VETEMENT. Inversement, on dit que la classe VETEMENT est **une sous-classe** de la classe ARTICLE. Pour comprendre le mécanisme d'héritage, on peut considérer qu'il s'agit ni plus ni moins que d'un mécanisme de copie automatique de texte source (vue 125). Tout se passe «comme si» toutes les primitives de la classe ARTICLE étaient recopiées dans la classe VETEMENT.



On peut même considérer que tout se passe comme si l'on venait d'écrire le texte source Eiffel des vues 126, 127, 128 et 129. Sur ces vues, les lignes recopiées automatiquement sont marquées d'une astérisque dans la colonne de gauche.

```

class VETEMENT
  creation {ANY}
    make
  * feature {ANY} -- Variables d'instances :
  *   designation : STRING
  *   prix_ht : REAL
  *   quantite : INTEGER
Vue 126 * feature {ANY} -- Constructeur :
  *   make (nom : STRING prix : REAL; nombre : INTEGER) is
  *     require
  *       prix > 0; nombre >= 0
  *     do
  *       designation := nom
  *       prix_ht := prix
  *       quantite := nombre

```

```

  *     ensure
  *       designation = nom; prix_ht = prix
  *       quantite = nombre
  *     end -- make
  * feature {ANY} -- Consultations :
  *   prix_ttc : REAL is
  *     do
  *       Result := prix_ht * 1.186
  *     end -- prix_ttc
Vue 127 * feature {ANY} -- Modifications :
  *   retirer (nombre : INTEGER) is
  *     require
  *       nombre > 0; nombre <= quantite
  *     do
  *       quantite := quantite - nombre

```

Vue 128

```

|*      ensure
|*      quantite = ( old quantite) - nombre
|*      end -- retirer
|*  ajouter (nombre : INTEGER) is
|*      require
|*      nombre > 0
|*      do
|*      quantite := quantite + nombre
|*      ensure
|*      quantite = (old quantite) + nombre
|*      end -- ajouter

```

Vue 129

```

feature {ANY}
  taille : INTEGER
  coloris : STRING
  solder (remise : REAL) is
    require
      0 < remise; remise < 100
    do
      prix_ht := prix_ht * (1 - (remise / 100))
    ensure
      prix_ht = (old prix_ht) * (1 - (remise / 100))
    end -- solder
|* invariant
|*  quantite >= 0; prix_ht >= 0
end -- class VETEMENT

```

**Exercice 45** Définir la classe VETEMENT sans utiliser l'héritage en procédant par composition. Comparez la solution obtenue avec celle faisant appel à l'héritage.

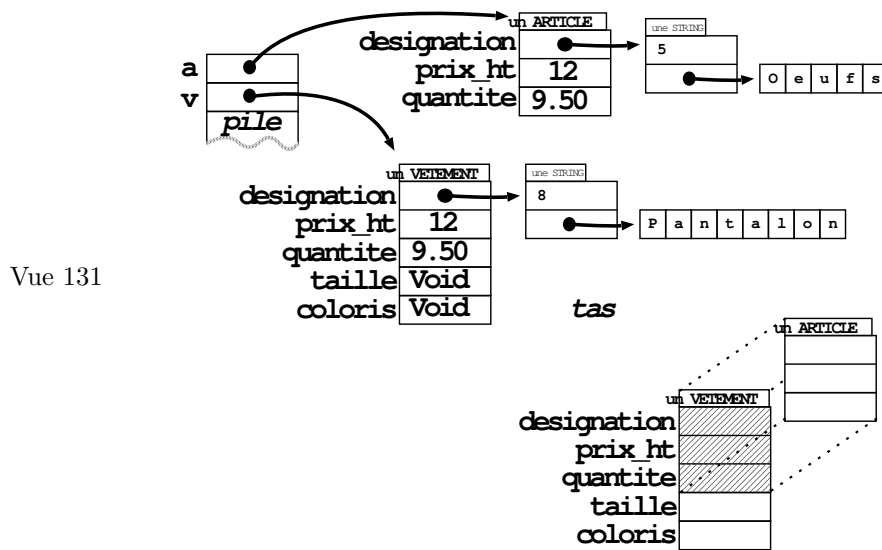
Toutes les primitives de la classe ARTICLE sont utilisables avec une variable de type VETEMENT mais l'inverse n'est pas vrai (vue 130). Une instance de la classe VETEMENT possède toutes les variables d'instances définies dans la classe ARTICLE ainsi que toutes les variables d'instances spécifiquement définies dans la classe VETEMENT (vue 131). Il est important (pour bien comprendre la suite) que le début d'une instance de la classe VETEMENT se «superpose» exactement sur une instance de la classe ARTICLE.

### Clients de la classe VETEMENT

```
local
  v : VETEMENT
  a : ARTICLE
do
  create a.make("Oeufs",9.50,12)
  create v.make("Pantalon",300.00,25)
  a.ajouter(10)
  v.ajouter(15)
  v.solder(10) ☉ ← Point d'arrêt ici.
```

Vue 130





Dans les méthodes d'une sous-classe, **toutes** les variables d'instances de ses superclasses sont accessibles. La relation d'héritage est transitive et, plus la classe est «basse» dans le graphe d'héritage, plus le nombre de variables d'instances augmente.

## 17 Premières règles d'héritage

## Premières règles de l'héritage

- primitives** : copiées par défaut avec le même niveau de visibilité (**feature** {*X*, *Y*, ...})
- Vue 132 **constructeurs** : la (les) clause(s) **creation** ne sont jamais copiées
- invariant** : toujours copié et considéré prioritairement avant l'invariant propre à la sous-classe

Le mécanisme d'héritage obéit à des règles bien précises. Tout le texte de la superclasse n'est pas systématiquement copié n'importe comment et n'importe où dans la sous-classe (vue 132).

### 17.1 Héritage des primitives

Par défaut, les primitives sont systématiquement copiées dans la sous-classe avec le même niveau de visibilité que celui de la classe d'origine. Une primitive sous couvert d'une clause **feature** {*X*} dans la superclasse se retrouve donc également sous couvert d'une clause **feature** {*X*} dans la sous-classe.

### 17.2 Héritage des constructeurs

La clause **creation** de la superclasse n'est jamais copiée dans la sous-classe. Si une sous-classe ne comporte pas de clause **creation**, seul le constructeur par défaut ("**create**") est donc disponible.

### 17.3 Héritage de l'invariant

L'invariant de la superclasse est toujours copié dans la sous-classe. Si la sous-classe comporte un invariant propre, tout se passe comme si celui de la superclasse était copié avant celui de la sous-classe.

## 18 Contrôler l'héritage

De nombreuses options dans la clause d'héritage permettent d'adapter les règles d'héritage.

## 18.1 Renommer une méthode héritée

Il est possible de renommer une méthode héritée en utilisant le mot clé **rename**. Sur l'exemple de la vue 133, la méthode **make** de la classe **ARTICLE** est renommée **make\_article** dans la classe **VETEMENT**. Ce faisant, il faut bien comprendre que seul le nom **make\_article** est utilisable pour invoquer la méthode **make** de la classe **ARTICLE**. Le nom **make**, maintenant «libre» dans la classe **VETEMENT** peut être utilisé pour donner une nouvelle définition du constructeur. A l'intérieur du corps de la nouvelle définition de **make**, on fait appel à la méthode **make** de la classe **ARTICLE**. Cette technique est courante en programmation par objets. On dit que l'on fait appel à la «**superméthode**» (voir aussi page 165).

```
class VETEMENT
inherit
  ARTICLE
  rename make as make_article
end
creation {ANY}
  make
feature {ANY}
  make (nom : STRING; prix : REAL; nombre : INTEGER
    mensuration : INTEGER; couleur : STRING) is
  require
    27 < mensuration; mensuration < 65
  do
    make_article (nom, prix, nombre)
    taille := mensuration
    coloris := couleur
```

Vue 133

Vue 134

```

ensure
  taille = mensuration; coloris = couleur
end -- make
taille : INTEGER
coloris : STRING
solder (remise : REAL) is
  require
    ...
  end -- solder
invariant
  taille >= 0
  27 < taille ; taille < 65
end -- class VETEMENT

```

```

create v1.make("Pantalon",299.50,25,36,"Vert")
v1.ajouter(10)
v1.solder(10)
v.make_article("Oeufs",9.50,12)

```

## Adapter les règles d'héritage

Vue 135

```

class VETEMENT
inherit
  ARTICLE
  rename make as make_article
  export {NONE} make_article
end
creation {ANY}
make
...

```

## 18.2 Changer le statut d'exportation

Une autre adaptation possible des règles d'héritage consiste à pouvoir intervenir sur le niveau d'exportation d'une primitive héritée. Ainsi, il est possible par exemple de changer le statut d'exportation de la primitive `make_article` (vue 135). Il est désormais interdit d'utiliser `make_article` sur une variable de type `VETEMENT`. Ceci ne remet pas en cause le reste de la classe car `make_article` reste applicable sur `Current` (comme d'habitude).

## 19 Spécialiser par masquage

### 19.1 Redéfinition d'une méthode héritée

Il est possible de changer la définition d'une primitive héritée en signalant son intention dans la clause `inherit` grâce au mot clé `redefine` (vue 136). La nouvelle classe obtenue, `ARTICLE_DE_LUXE` a donc exactement la même interface que sa superclasse `ARTICLE` (vue 137). Seul «l'effet» de la méthode `prix_ttc` est différent : la taxe appliquée aux `ARTICLE_DE_LUXE` est de 33%.

### Spécialiser par masquage

```

class ARTICLE_DE_LUXE
inherit
  ARTICLE
  redefine prix_ttc
end
Vue 136 creation {ANY}
make
feature {ANY}
  prix_ttc : REAL is
  do
    Result := prix_ht * 1.33
  end -- prix_ttc
end -- class ARTICLE_DE_LUXE

```

## Spécialiser par masquage

```
local
  a : ARTICLE
  adl : ARTICLE_DE_LUXE
do
  create a.make("Pain",1.00,100)
  create adl.make("Caviar",100.00,1)
  a.retirer(100)
  adl.retirer(1)
  io.put_real(a.prix_ttc) -- ← imprime 1.186
  io.put_real(adl.prix_ttc) -- ← imprime 133.00
```

Vue 137

Le profil d'une primitive redéfinie doit être conforme au profil de la primitive héritée correspondante. Par exemple, le nombre d'arguments d'une routine ne doit pas changer dans la sous-classe.

### 19.2 Héritage des assertions

Si l'on souhaite revenir sur les assertions d'une méthode qui est redéfinie, il faut utiliser la syntaxe **require else** et **ensure then** (vue 138). Ces mots clés sont là afin de rappeler l'ordre dans lequel les pré- et post- conditions doivent être évaluées.

## Héritage des assertions

Vue 138

```

class B
inherit A
  redefine m
  end
...
m ... is
  require else
    ... -- ← pré-condition plus faible
  do
    ...
  ensure then
    ... -- ← post-condition plus forte
  end -- m

```

## Exécution des assertions

Vue 139

```

require
   $pré(m_B)$  or else  $pré(m_A)$ 
ensure
   $post(m_B)$  and then  $post(m_A)$ 

```

## 20 Polymorphisme de variables et liaison dynamique

### 20.1 Polymorphisme de variable

En mémoire, le début d'une instance de la classe VETEMENT est complètement identique à une instance de la classe ARTICLE. Toutes les méthodes de la classe ARTICLE peuvent donc s'appliquer sur les instances de la classe VETEMENT. La réciproque n'est pas vraie et, pour cette raison, un seul sens d'affectation est autorisé entre les variables déclarées ARTICLE et les variables déclarées VETEMENT (vue 140).

Une même variable peut donc, au cours de l'exécution d'un programme, référencer des instances de classes différentes.

Pour une variable, on parle soit de **type statique** pour indiquer la classe de déclaration de la variable, ou on parle de **type dynamique** pour désigner à un instant donné la classe d'appartenance de l'instance référencée par la variable.

Le compilateur ne se préoccupe que du type statique pour effectuer les vérifications sémantiques. La vue 141 donne les règles de concordances de types.

La notation **like** permet de déclarer une variable en faisant référence à une autre variable, ou à une fonction ou encore par rapport à **Current** (vue 142).

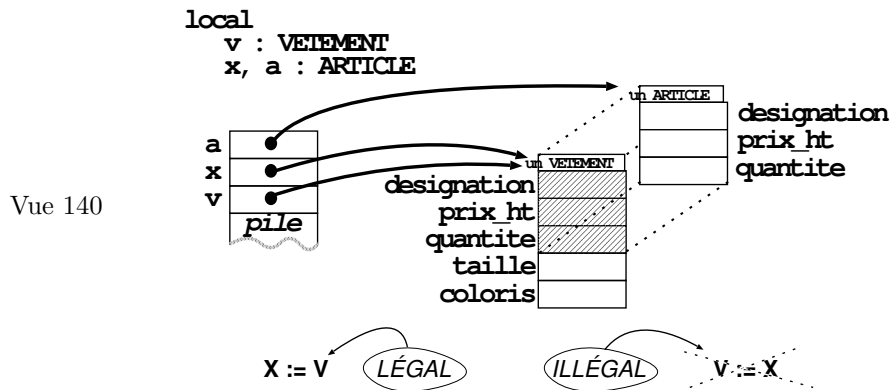
Considérons l'exemple d'héritage présenté sur la vue 143. Que se passe-t-il dans la classe RECTANGLE si l'on suppose que la classe POLYGONE contient les définitions présentées sur la vue 144 ?

Dans la classe RECTANGLE, tout se passe **comme si** le texte des méthodes `intersection1` et `intersection2` était recopié. La déclaration **like Current** est donc interprétée de façon spécifique, après recopie, dans la classe RECTANGLE (cf. vue 145 et 146).

**Exercice 46** Dressez la liste des appels acceptés par le compilateur pour la méthode `intersection3` ajoutée dans la classe POLYGONE avec le profil :

```
intersection3 (p2 : like Current): RECTANGLE is
  do ...
  end -- intersection3
```





## Concordance de types

Un type Y concorde avec un type X si :

- X et Y identiques
- X n'est pas une classe générique et Y hérite de X
- X et Y sont des classes génériques et Y hérite de X et chaque paramètre générique de Y concorde avec le paramètre générique correspondant de X
- Y est de la forme **like** repère, et repère concorde avec X
- Y concorde avec un type Z et Z concorde avec Y.

Vue 141

## Typage statique

Vue 142

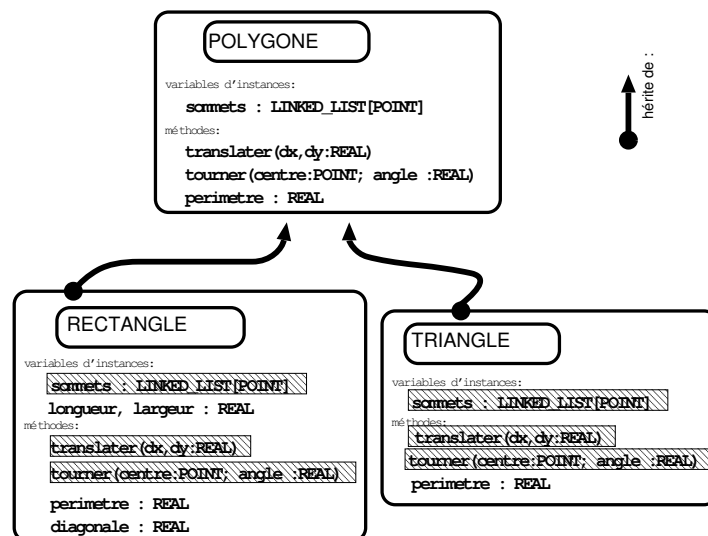
```

v : VETEMENT
v2 : like v
x : PILE[INTEGER]
y : like x
...
f1 (arg1 : INTEGER) : REAL is ...
f2 (arg1 : like f1) is ...
f3 (arg1 : like Current) is ...
t : like Current

```

Indiquer un type statique avec la mention **like** revient à fixer «relativement» le type statique.

Vue 143



Vue 144

```

class POLYGONE
feature {ANY}
  intersection1 (p2 : POLYGONE): POLYGONE is
    do ...
    end -- intersection1
  intersection2 (p2 : like Current): like Current is
    do ...
    end -- intersection2
  ...
end -- class POLYGONE

class RECTANGLE
inherit POLYGONE
  ...
end -- class RECTANGLE

```

## Vérifications sémantiques statiques

Vue 145

```

...
p1, p2, p3 : POLYGONE
r1, r2, r3 : RECTANGLE
do
  p1 := p2.intersection1(p3)
  p1 := p2.intersection1(r3)
  p1 := r2.intersection1(p3)
  p1 := r3.intersection1(r3)
  ...

```

**Exercice :** parmi les  $2^3 * 2$  possibilités d'appels de `intersection1` et `intersection2`, combien y-a-t-il d'appels sémantiquement corrects ?

Vue 146

```

-- Pour intersection1 dans POLYGONE:
-- POLYGONE × POLYGONE → POLYGONE
p1 := p2.intersection1(p3)
p1 := p2.intersection1(r3)
-- Pour intersection2 dans POLYGONE:
-- POLYGONE × POLYGONE → POLYGONE
p1 := p2.intersection2(p3)
p1 := p2.intersection2(r3)
-- Pour intersection1 dans RECTANGLE:
-- RECTANGLE × POLYGONE → POLYGONE
p1 := r2.intersection1(p3)
p1 := r2.intersection1(r3)
-- Pour intersection2 dans RECTANGLE:
-- RECTANGLE × RECTANGLE → RECTANGLE
r1 := r2.intersection2(r3)
p1 := r2.intersection2(r3)
...

```

## 20.2 Liaison dynamique

Une question essentielle se pose lorsque qu'il y a masquage d'une méthode. Quelle est la méthode invoquée ? celle qui correspond au type statique de la variable ? ou bien celle qui correspond au type dynamique ? Réponse : celle correspondant au type dynamique bien sûr (vue 147). Finalement, si la variable `article` référence une instance de la classe `ARTICLE`, la TVA calculée est de 18% et si la même variable référence une instance de la classe `ARTICLE_DE_LUXE`, la TVA calculée passe automatiquement à 33%.

En indiquant le nom de la classe à instancier entre les deux points d'exclamations qui marquent l'instanciation, il est possible d'éviter l'utilisation d'une variable intermédiaire pour instancier par exemple un `VETEMENT` et faire en sorte qu'il soit référencé par une variable de type statique `ARTICLE` (vue 148).

## Liaison dynamique

Priorité au type dynamique pour sélectionner la méthode

```

article : ARTICLE
unLuxe : ARTICLE_DE_LUXE
do
  if age_du_capitaine = 33 then
    create unLuxe.make("Caviar",100.00,1)
    article := unLuxe
  else
    create article.make("Oeufs",9.50,12)
  end
  io.putreal(article.prix_ttc) ← ICI
  ...

```

Vue 147

## Indiquer la classe à instancier

Priorité au type dynamique pour sélectionner la méthode

```

article : ARTICLE
do
  if age_du_capitaine = 33 then
    create {ARTICLE_DE_LUXE} article.make("Caviar",100.00,1)
  elseif vitesse_du_vent = 12 then
    create {VETEMENT} article.make("Pantalon",299.50,25,36,"Vert")
  else
    create article.make("Oeufs",9.50,12)
  end
  ...

```

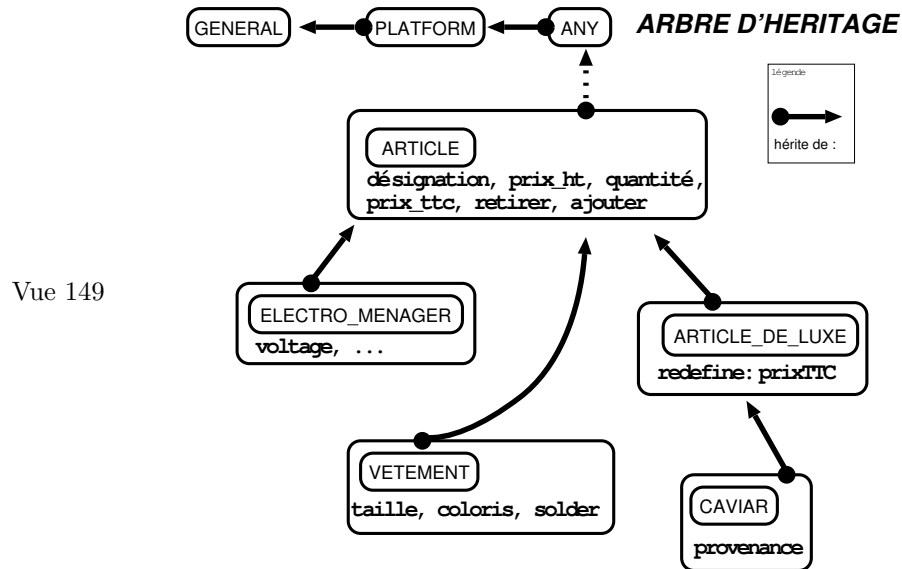
Vue 148

**Exercice 47** Écrire une classe `COMMANDE` possédant une opération de calcul du montant total TTC d'une commande. Utilisez la liaison dynamique.

## 21 Relation d'héritage

### 21.1 L'arbre d'héritage

La relation d'héritage est transitive. On peut imaginer de classer les diverses catégories d'articles de la même manière que l'on classe le règne animal (vue 149). Même si elle n'est pas mentionnée, par défaut, toute classe hérite de la classe `ANY`. La classe `GENERAL` est la racine de l'arbre d'héritage.



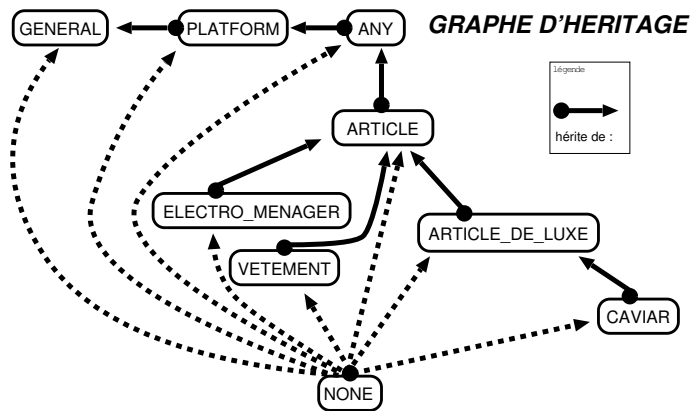
```
class GENERAL
...
  frozen io: STD_INPUT_OUTPUT is
    -- Handle to standard file setup.
    -- To use the standard input/output file.
    once
      create Result.make
    ensure
      Result /= Void
    end
  frozen Void: NONE is
    -- Void reference.
    external "SmartEiffel"
    end

  frozen clone(other : ANY) : like other is
    -- Void if 'other' is Void; otherwise new object
    -- equal to 'other'.
    do
      if other /= Void then
        Result := other.twin
      end
    ensure
      equal: equal(Result,other)
    end
end -- class GENERAL
```

Vue 150

Vue 151

Vue 152



Vue 153

```

class PLATFORM
inherit
    GENERAL
end
...
Maximum_character_code : INTEGER is 255
...
end -- class PLATFORM

class ANY
inherit
    PLATFORM
end
end -- class ANY
    
```



Il est possible de donner deux noms à une même primitive. En outre, il est possible également d'utiliser le mot clé **frozen** (Vue 150) devant un nom de primitive. Cette précision a pour effet de «geler» la définition pour le nom en question, c'est à dire d'éviter toute possibilité de redéfinition dans une sous-classe. Sur l'exemple, comme le mot clé **frozen** se trouve devant le nom `standard_is_equal` il est désormais interdit de donner dans toutes les sous-classes de `GENERAL` une nouvelle définition portant ce nom. Dans toutes les classes donc, le nom `standard_is_equal` fait référence à la définition qui est dans la classe `GENERAL`.

La classe `ANY` peut être redéfinie et il est possible par exemple d'y ajouter des primitives d'intérêt général comme par exemple l'accès aux entrées sorties standards (vue 154). Il faut cependant rester très prudent chaque fois que l'on change la définition de la classe `ANY` car la modification de cette classe a des conséquences sur l'ensemble des classes de la bibliothèque.

Ne mettez pas n'importe quoi dans la classe `ANY`. Par exemple, si on ajoute dans la classe `ANY` la définition d'une variable d'instance, on ajoute également cette variable d'instance (par transitivité de la relation d'héritage), dans toutes les classes. Inutile de dire que c'est la catastrophe du point de vue de la mémoire occupée par les instances.

### Redéfinition possible de la classe `ANY`

```

class ANY
inherit
  PLATFORM
end
Vue 154 feature {NONE}

  danger : PAS_CA

end -- class ANY

```

### PRUDENCE !

**Exercice 48** Ajouter une post-condition à la méthode périmètre de la classe rectangle (de la vue 143) pour vérifier que le résultat calculé est bien identique à celui calculé par la superméthode.

L'opérateur d'affectation noté " ?=" permet de faire une tentative d'affectation non conforme. On peut, en utilisant cet opérateur, essayer d'aller *régulièrement* à l'encontre des règles de concordance de types (énoncées vue 141). S'il apparaît à l'exécution que le type dynamique est finalement conforme, l'affectation est effectuée sinon, la variable est mise à `Void`. Par exemple, la notation `?=` permet d'essayer de faire une affectation dans le sens :

```
vetement ?= article ;
```

Deux cas de figures sont possibles à l'exécution. Soit le type dynamique de la variable `article` est `VETEMENT` et dans ce cas l'affectation est faite. Soit le type dynamique de la variable `article` n'est pas conforme avec le type statique de la variable `vetement` : il ne s'agit ni d'une instance de la classe `VETEMENT` ni d'une instance d'une sous-classe de `VETEMENT` (c'est par exemple une instance de la classe `ARTICLE_DE_LUXE`). Dans ce deuxième cas, la variable `vetement` est mise à `Void`. Il va sans dire que si l'on peut **éviter d'utiliser l'opérateur `?=`** c'est mieux.

**Exercice 49** Avec l'opérateur `?=`, il est possible de «simuler» la généricité en utilisant l'héritage. Écrire une classe `PILE`, non générique, permettant d'empiler toutes sortes d'instances. Dressez la liste des avantages et des (gros) inconvénients de cette solution.

## Opérateur `?=`

*Tentative d'affectation potentiellement non conforme !*

Vue 155

```

tab_any : ARRAY[ANY]
a : ARTICLE
v : VETEMENT
adl : ARTICLE_DE_LUXE
p : POINT
i : INTEGER
do
  create a.make("Oeufs",9.50,12)
  create adl.make("Caviar",100.00,2)
  create v.make("Pantalon",299.50,25,36,"Vert")
  create p.make(1,1)
tab_any := <<a,adl,v,p>>

```

Vue 156

```

from
  i := tab_any.lower
until
  i > tab_any.upper
loop
  a ?= tab_any.item(i)
  if a /= Void then
    io.put_real(a.prix_ttc)
  else
    p ?= tab_any.item(i)
    io.put_real(p.x)
  end
  io.put_newline
  i := i + 1
end

```

## 22 Hériter pour implanter

Parmi les utilisations possibles du mécanisme d'héritage, il en est une qu'il faut absolument présenter car elle est naturellement (et pernicieusement) alléchante. L'idée est simple : une pile est composée entre autres d'un tableau (cf. classe PILE de la vue 114) de la classe ARRAY, pourquoi ne pas hériter de ladite classe ARRAY pour implanter la classe PILE ? Après tout, si l'on n'y regarde pas de trop près, une pile *est une sorte de* tableau !

Cette idée est mise en œuvre sur les vues 157, 158, 159 et 160 qui donnent la définition de la classe PILE\_MOCHE.

Bien sûr, une telle pile peut être utilisée exactement de la même façon que l'on utilise une pile instance de la classe PILE de la vue 114. Partout où l'on utilise la classe PILE (comme sur la vue 118 par exemple), on peut également utiliser la classe PILE\_MOCHE. Cependant, comme cette classe hérite de la classe ARRAY, il faut bien être conscient du fait que toutes les opérations de ARRAY sont héritées (vue 161). On peut donc faire n'importe quoi dans une PILE\_MOCHE.

*Une pile est une sorte de tableau ...*

## Hériter pour implanter

Vue 157

```
class PILE.MOCHE[T]
inherit
  ARRAY[T]
  rename make as make_array
  export {NONE} make_array
end
creation {ANY}
  make
feature {ANY}
  nb_elements : INTEGER
```

Vue 158

```
feature {ANY} -- Constructeur :
  make is
    do
      make_array(1,2)
      nb_elements := 0
    ensure
      vide
    end -- make
feature {ANY} -- Consultations :
  vide : BOOLEAN is
    do
      Result := (nb_elements = lower - 1)
    end -- vide
```

Vue 159

```

    sommet : T is
      require
        not vide
      do
        Result := item(nb_elements)
      end -- sommet
  feature {ANY} -- Modifications :
    empiler (x : T) is
      do
        nb_elements := nb_elements + 1
        force(x,nb_elements)
      ensure
        not vide
        sommet = x
        nb_elements = ( old nb_elements) + 1
      end -- empiler

```

Vue 160

```

    depiler is
      require
        not vide
      do
        nb_elements := nb_elements - 1
      ensure
        nb_elements = ( old nb_elements) - 1
      end -- depiler
  invariant
    nb_elements >= 0
    upper >= nb_elements
end -- class PILE.MOCHE

```

*Une PILE\_MOCHE est une PILE :*

```

local
  pm : PILE_MOCHE[INTEGER]
do
  create pm.make
  pm.empiler(1)
  io.put_int(pm.sommet)

```

Vue 161

*Une PILE\_MOCHE est un ARRAY :*

```

pm.put(0,1)
pm.reindex(2)

```

**Une PILE\_MOCHE est moche**

## 22.1 Cohérence locale – Cohérence globale

Même si on pense à changer le statut d’exportation des opérations qui ne sont pas souhaitables (vue 162), il est toujours possible de «bricoler» directement la pile en utilisant une variable de type statique ARRAY (vue 163).

L’exemple d’utilisation de la classe PILE\_MOCHE de la vue 163 est un exemple «d’incohérence globale» d’un programme Eiffel. Pourquoi dit-on qu’il y a incohérence «globale» et non pas incohérence tout court ? Réponse : chaque instruction de la vue 163, prise **une à une** est parfaitement légale. On dit dans ce cas que le programme respecte la «cohérence locale». Et pourtant, force est de constater qu’il est possible dynamiquement, d’appliquer sur une instance de la classe PILE\_MOCHE des opérations statiquement non exportées. Ceci explique l’utilisation du terme «incohérence globale».

Une notation spéciale dans la clause **export** permet d’éviter la longue énumération de primitives de la vue 162. La nouvelle version de la classe PILE\_MOCHE de la vue 164 utilise la notation en question. Nota : cela ne règle en rien le problème d’incohérence globale.

Je n’ai même pas essayé de définir une version de la classe PILE\_TRES\_MOCHE, héritant à la fois de la classe ARRAY pour avoir le tableau et héritant aussi de la classe INTEGER pour avoir l’indice.

La redéfinition d’une primitive peut elle aussi amener à une situation d’incohérence globale (vue 165). Ce genre de vérifications (pas si évidentes que ça !) sont rarement effectuées par les compilateurs du commerce.

Vue 162

```
class PILE_MOINS_MOCHE[T]
inherit
  ARRAY[T]
  rename make as make_array
  export {NONE} make_array, resize, reindex,
    clear_all, wipe_out, put, force, insert,
    remove, move, lower, upper, count, size,
    empty, all_cleared, to_external
end
...
```

Vue 163

```
local
  pm : PILE_MOINS_MOCHE[INTEGER]
  a : ARRAY[INTEGER]
do
  ...
  if vitesse_du_vent = 3 then
    a := pm
  else
    create a.make(2,3)
  end
  a.put(0,1)
  a.reindex(2)
```

**Incohérence globale**

*Le mot clé **all** dans la clause export permet d'éviter une longue énumération de primitives.*

```

class PILE_MOINS_MOCHE[T]
inherit
  ARRAY[T]
  rename make as make_array
  export {NONE} all
end
...

```

Vue 164

*ce qui ne règle pas le problème d'incohérence globale de la vue 163*

### Redéfinition et cohérence globale

```

class ALPHA |class GAMMA
feature |feature
end -- class ALPHA | test: ALPHA
-----| create is
class BETA | do
inherit ALPHA | create test
creation | end
fred |end -- class GAMMA
feature |-----
fred is |class DELTA
do -- ... |inherit GAMMA redefine test end
end |feature
end -- class BETA | test: BETA
|end -- class DELTA

```

Vue 165



## 23 Classe abstraite

La notion de classe abstraite est à rapprocher de celle de type abstrait algébrique. Une classe abstraite ne peut pas être instanciée et certaines primitives ne sont pas définies (ces primitives sont en quelque sorte l'équivalent des constructeurs dans un type abstrait algébrique). Syntactiquement, il suffit de faire précéder la définition de la classe par le mot clé **deferred**. Le même mot clé est utilisé pour les primitives que l'on ne souhaite pas définir en remplacement de la partie **do**. Les vues 166 et 167 montrent la classe abstraite `COMPARABLE` extraite de la bibliothèque.

Il est donc interdit d'instancier la classe `COMPARABLE`. Cette classe doit être utilisée par le biais de l'héritage, la sous-classe étant chargée de définir au moins l'opération **infix** `"<"` dont l'implantation a été retardée. C'est ce qui est fait dans la classe `STRING` de la bibliothèque (vue 168). Dans cette classe, la fonction `compare` est aussi redéfinie pour des raisons d'efficacité évidentes.

De nombreuses autres classes de la bibliothèque héritent directement ou indirectement de la classe `COMPARABLE` comme par exemple `INTEGER`, `REAL` ou `CHARACTER`. L'intérêt d'une classe abstraite est donc évident : mettre en facteur des algorithmes communs. Si vous définissez une nouvelle classe d'objet sur lesquels il existe un ordre total, il convient donc d'hériter de la classe `COMPARABLE`.

Si on hérite d'une classe abstraite (**deferred**), cette classe reste abstraite (même sans le mot clé **deferred**) si elle ne donne pas une définition à **toutes** les primitives dont l'implantation a été retardée (**deferred**). De toute évidence, une classe abstraite ne doit pas avoir de constructeurs (pas de clause de **creation**).

```

deferred class COMPARABLE
feature {ANY}
  infix "<" (other : like Current) : BOOLEAN is
    require
      other_not_void : other /= Void
    deferred
    ensure
      anti_symmetric: Result implies not (other < Current)
    end
  infix "<=" (other : like Current) : BOOLEAN is
    require
      other_not_void : other /= Void
    do
      Result := not (other < Current)
    end

```

Vue 166

Vue 167

```

infix ">" (other : like Current) : BOOLEAN is
  require ... do ... end
infix ">=" (other : like Current) : BOOLEAN is
  require ... do ... end
compare (other : like Current) : INTEGER is
  -- Compare Current with 'other'.
  -- '<' <=> Result < 0
  -- '>' <=> Result > 0
  -- Otherwise Result = 0
  require ... do ... end
end -- class COMPARABLE

```

COMPARABLE est une classe ABSTRAITE

Vue 168

```

class STRING
inherit
  COMPARABLE
  redefine compare
end
....
infix "<" (other : STRING) : BOOLEAN is
  do ... end
compare (other : STRING) : INTEGER is
  do ... end
...
end -- class STRING
-----
local
  s : STRING
do
  if (s < "toto") or (s >= "abc") then ...

```

## 24 Généricité contrainte

Dans certains cas, on souhaite pouvoir définir une classe en la paramétrant par une autre classe (comme dans le cas de la classe `PILE[T]` de la vue 114). Cependant, on est souvent obligé de faire certaines suppositions sur les opérations du type en paramètre : les éléments sont-ils ou non comparables ? possèdent-ils ou non une opération d'impression ? etc.

Par exemple, si on souhaite écrire une classe capable de gérer des ensembles de n'importe quoi, il faut être en mesure de comparer les éléments avec autre chose que l'opérateur de comparaison physique (`=`). Or, avec la généricité non contrainte, seul cet opérateur est utilisable sur les objets du type paramètre.

La notation Eiffel pour la généricité contrainte évite l'énumération des opérations que l'on suppose avoir dans la classe paramètre. Les opérations supposées présentes sont mentionnées par le biais de l'héritage (vue 169).

Dans la liste des paramètres génériques formels, la classe à droite de la flèche, "`->`", doit effectivement exister. Toutes les opérations de cette classe peuvent être utilisées dans la classe générique pour manipuler les objets du type paramètre.

Au moment de la déclaration d'une variable de la classe `SORTED_LIST` (c'est à dire au moment de l'instanciation de la classe générique), le compilateur vérifie que le paramètre générique effectif est bien une classe qui hérite de la classe `COMPARABLE`.

La vue 170 montre comment on utilise la classe `SORTED_LIST`. Au moment de l'instanciation, le paramètre booléen du constructeur `make` indique que l'on ne veut pas mettre plusieurs occurrences d'un même élément (au sens `is_equal` du terme) dans la liste triée. Au fur et à mesure des adjonctions, les éléments sont insérés en bonne place relativement à la relation d'ordre effectivement implantée dans la classe correspondant au paramètre générique effectif (`STRING` dans le cas de l'exemple). La liste est constamment (et automatiquement) maintenue triée.

### Généricité contrainte

*Définition d'une classe qui maintient automatiquement triée une liste d'éléments comparables.*

```

class SORTED_LIST [G -> COMPARABLE]
  ...
  if unG < Current.item(i) then
  ...
end -- class SORTED_LIST

```

Vue 169

G : paramètre générique formel

COMPARABLE : classe existante fixant les opérations utilisables sur les objets de la classe G

Vue 170

```
local
  copains : SORTED_LIST[STRING]
  i: INTEGER
do
  create copains.make(True)
  copains.add("raymond")
  copains.add("albert")
  copains.add("lucien")
  from -- Impression dans l'ordre alphabétique :
    i := 1
  until
    i > copains.upper
  loop
    io.put_string(copains.item(i))
    io.put_newline
    i := i + 1
  end
```

## 24.1 Limitation

Il est interdit d'instancier le paramètre générique formel d'une classe générique (vue 171). Ce qui est cohérent avec le choix (discutable) de lier généricité et héritage.

Limitation : *Il est interdit d'instancier la classe désignée par le paramètre générique formel.*

```

class POLYNOME[K -> COEFF]
...
local
  k : K
do
  create k...
  ↑ INTERDICTION :-()
...
end -- class POLYNOME

```

Vue 171

**Exercice 50** Construire l'interface d'une classe abstraite de manipulation de MATRICES. Implanter ensuite de diverses façons cette classe abstraite en mettant le plus possible d'opérations en facteur au niveau de la classe MATRICE.

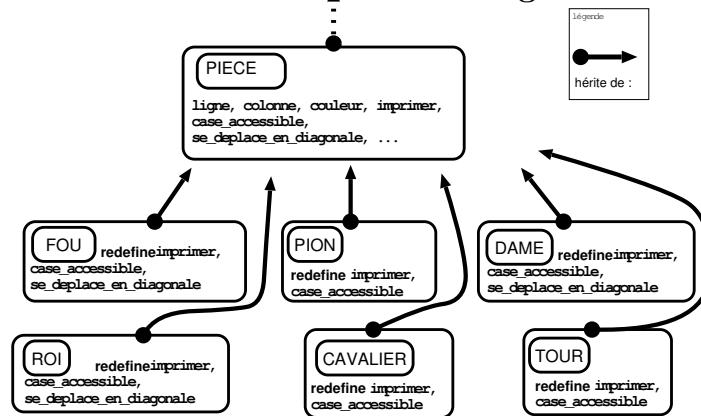
## 25 Classifier en utilisant l'héritage

Dès le premier moment de la conception d'un logiciel, sachant que l'on va manipuler des objets ayant un bon nombre de caractéristiques communes, on peut songer à introduire une entité abstraite pour mettre en facteur ces propriétés communes. Ce qui est spécifique à chacune des catégories sera traité dans des sous-classes différentes.

Par exemple, dans le cadre de la réalisation d'un jeu d'échecs, on peut décider de la classification donnée sur la figure 172.

## Classier avec l'héritage

Vue 172



Vue 173

```

...
    unePiece : PIECE
    uneCase : CASE
...
    if unePiece.se_deplace_en_diagonale then
...
        if unePiece.case_accessible(uneCase) then
...
            unePiece.imprimer
...
  
```

*La liaison dynamique évite d'écrire explicitement les tests concernant la catégorie de pièce manipulée*

L'objectif d'une classification est de pouvoir éviter l'écriture de conditionnelles en utilisant la liaison dynamique. Supposons par exemple qu'au jeu d'échecs, seuls les fous, les rois et les reines se déplacent en diagonale. Il est possible d'implanter le prédicat `se_deplace_en_diagonale` sans jamais écrire une conditionnelle (vues 174 et 175).

```
deferred class PIECE
feature {ANY}
  se_deplace_en_diagonale : BOOLEAN is
    deferred
  end -- se_deplace_en_diagonale
...
end -- class PIECE
```

Vue 174

```
-----
class FOU
inherit PIECE
feature {ANY}
  se_deplace_en_diagonale: BOOLEAN is True
...
end -- class FOU
```

```
class TOUR
inherit PIECE
feature {ANY}
  se_deplace_en_diagonale: BOOLEAN is False
...
end -- class TOUR
-----
class TOUR -- VARIANTE
inherit PIECE
feature {ANY}
  se_deplace_en_diagonale : BOOLEAN is
  do
    Result := False -- Instruction inutile :-()
  end
...
end -- class TOUR
```

Vue 175

Un variante consiste à utiliser la redéfinition (**redefine**) de la primitive `se_deplace_en_diagonale`. On place la définition la plus courante au niveau de la classe `PIECE` (vue 176 et 177).



```
deferred class PIECE
feature {ANY}
  se_deplace_en_diagonale : BOOLEAN is
  do
    -- False est le cas le plus courant.
  end -- se_deplace_en_diagonale
...
end -- class piece
-----

class TOUR
inherit PIECE
feature {ANY}
...
end -- class tour

class FOU
inherit
  PIECE
  redefine se_deplace_en_diagonale
end
feature {ANY}
  se_deplace_en_diagonale: BOOLEAN is True
...
end -- class fou
```

Vue 176

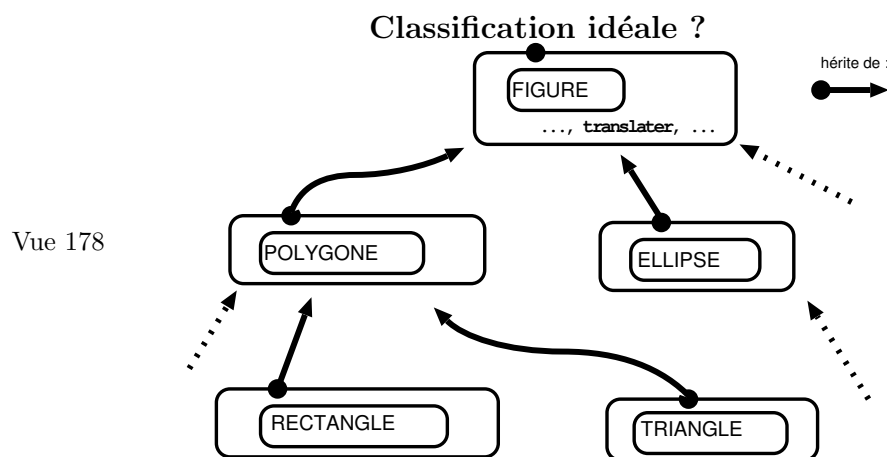
Vue 177

**Exercice 51** Donnez une définition du prédicat `se_deplace_en_diagonale` en utilisant l'opérateur `?=`.

**Exercice 52** Imaginez les classes à mettre en œuvre pour la construction d'un éditeur interactif de figures composées d'éléments géométriques simples (triangles, rectangles, points, cercles, traits, ...). Quel est l'intérêt de la liaison dynamique dans ce cas ?

**Exercice 53** Envisagez l'ajout d'un nouvel élément géométrique pouvant faire partie d'une figure. Quelles sont les modifications à effectuer ?

Il existe rarement une classification hiérarchique idéale (cf. exemple de la classification des figures géométriques de la vue 178).



## 26 Héritage multiple

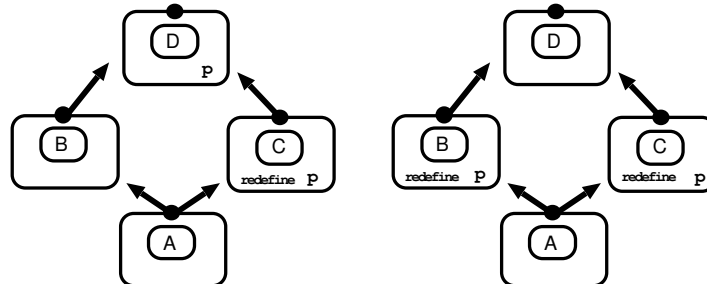
Certains langages de classes, dont Eiffel, autorisent une classe à avoir plusieurs super-classes directes. On dit dans ce cas qu'il y a possibilité d'héritage multiple.

Le problème des conflits d'héritage se pose immédiatement (vue 179).

La syntaxe de la clause **inherit** est relativement complexe (vue 181) et la compréhension de toutes les options n'est pas toujours facile ! Dans la mesure où l'on a la maîtrise complète d'une partie séparée du graphe d'héritage, il est à mon avis toujours possible d'éviter ces options complexes en procédant par factorisation et renommage manuels.

**Héritage multiple** : une classe possède plusieurs superclasses directes.

Vue 179



**Conflit** : dans la classe A car on ne sait pas laquelle des deux primitives *p* il faut choisir.

### Résolution d'un conflit d'héritage

Vue 180

```

class A
inherit
  B undefine p
end
  C
end

```

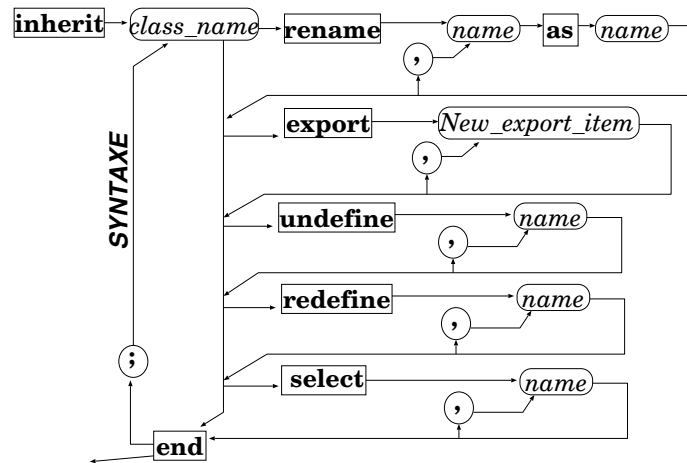
```

...
end -- class A

```

Pour choisir la primitive *p* de la classe C.

Vue 181



## 27 Comparer et dupliquer des objets

### 27.1 Comparer avec `equal`/`is_equal`

La définition de la fonction `equal` est gelée dans `GENERAL` (cf. définition théorique de la vue 182). Un problème se pose dans le cas où l'on souhaite comparer deux représentations différentes d'une même abstraction. Par exemple, il est raisonnable d'espérer un résultat vrai si l'on compare (en utilisant `equal`) deux collections contenant chacune le même nombre d'éléments, sachant que les éléments en question sont eux mêmes identiques deux à deux. La solution préconisée consiste à donner une nouvelle définition de la méthode `is_equal` dans la classe `COLLECTION` (vues 183 et 184). De manière indirecte, cette redéfinition influe sur le résultat donné par la fonction prédéfinie `equal`.

Lorsqu'une classe est bien conçue, sauf exception, on n'utilise jamais la fonction `deep_equal`.

**Exercice 54** Faire en sorte que la fonction prédéfinie `equal` permette la comparaison de polynômes indépendamment de la représentation choisie (polynôme creux, polynôme dense).

### 27.2 Dupliquer avec `clone`/`twin`/`copy`

Le même genre de problème se pose en ce qui concerne la duplication d'objets. Par exemple, suite à un `clone` sur un `TRIANGLE`, on peut souhaiter obtenir un triangle complètement indépendant du modèle de départ. La solution à retenir est du même ordre que précédemment. Même si la définition de `clone` est gelée (vue 185), il est possible d'influencer le résultat de cette fonction prédéfinie par redéfinition de la méthode `copy` (vue 186).

Remarque : il est impossible de donner une version complètement écrite en Eiffel de la fonction prédéfinie `clone` (vue 185). Cette fonction doit malheureusement être traitée de manière spécifique par le compilateur.

Finalement, lorsqu'une classe est bien conçue, il n'est jamais nécessaire d'utiliser les primitives `deep_*`. Par exemple, la classe `STRING` possède une définition propre des méthodes `copy` et `is_equal` (vue 187).

**Exercice 55** Redéfinir la méthode `copy` pour les différentes représentations de polynômes.

## Comparer deux objets

```
class GENERAL
  ...
  feature {NONE}
  frozen equal(o1: ANY; o2: ANY):BOOLEAN is
    -- S'agit-il d'objets identiques ?
  do
    Result :=
      (o1 = Void and o2 = Void) or
      (o1 /= Void and o2 /= Void
       and then o1.is_equal(o2))
  end
  ...
```

Vue 182

*Modifier le comportement de la fonction `equal`.*

## Ne jamais utiliser `deep_equal` !

```

deferred class COLLECTION[G]
inherit ANY redefine is_equal end
feature {ANY}
  is_equal(other: COLLECTION[G]): BOOLEAN is
    local
      i1, i2: INTEGER
    do
      if other.count = count then

from
      Result := True
      i1 := lower
      i2 := other.lower
    until
      i1 > upper
    loop
      if equal(item(i1),other.item(i2)) then
        i1 := i1 + 1
        i2 := i2 + 2
      else
        Result := False
        i1 := upper + 1
      end
    end
  end
end
end
end

```

Vue 183

Vue 184

## Dupliquer un objet

Vue 185

```

class GENERAL
  ...
  feature {NONE}
    frozen clone(other: ANY): like other is
      -- When argument 'other' is Void, return Void
      -- otherwise return 'other.twin'.
      do
        if other /= Void then
          Result := other.twin
        end
      end
    ensure
      equal(Result,o1)
    end

    frozen twin: like Current is
      external "SmartEiffel"
    end

```

Vue 186

```

class TRIANGLE
  inherit
    ANY redefine copy end
  ...
  feature {ANY}
    copy(t2: TRIANGLE) is
      do
        p1 := t2.p1.twin
        p2 := t2.p2.twin
        p3 := t2.p3.twin
      end
    ...

```

**Ne jamais utiliser clone/deep\_clone/deep\_copy !**

```

class STRING
inherit
  COMPARABLE
  redefine
    is_equal, infix "<", compare
end
  HASHABLE
  redefine
    is_equal, hash_code
end
...

```

Vue 187

## 28 Concevoir des programmes efficaces

L'efficacité d'un programme dépend essentiellement de la complexité des algorithmes mis en œuvre. Ceci étant, pour finaliser une application réellement efficace, il me semble important de maîtriser dès la phase de conception deux aspects essentiels : l'utilisation possible de l'aliasing ainsi que, de manière plus générale, la consommation mémoire du système. En effet, avec les opérations d'entrées/sorties, l'allocation dynamique de mémoire (i.e. la création d'objets dans le tas) est l'opération élémentaire qui est de loin la plus coûteuse. Il est donc très important de minimiser les allocations dans le tas<sup>1</sup>.

### 28.1 Utilisation de l'aliasing

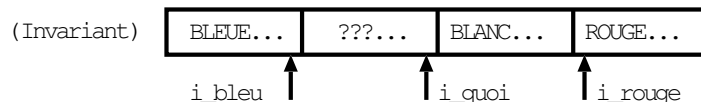
L'aliasing est le fait de désigner un unique et même objet par plusieurs chemins (plusieurs *alias*) différents. Ce phénomène se produit par exemple, si deux variables différentes référencent le même objet, ou, par exemple, si plusieurs cases d'un même tableau référencent le même objet. Les vues 188, 189 et 190 montrent comment il est possible d'utiliser le phénomène d'aliasing pour optimiser l'implantation de l'algorithme du drapeau tricolore. Rappelons que le problème consiste à trier un tableau contenant trois couleurs différentes à l'aide d'une seule itération ayant autant de pas (i.e. autant de répétitions) que le nombre d'éléments du tableau. La vue 188 montre l'initialisation de l'algorithme par rapport à

<sup>1</sup>Les allocations dans la pile ne coûtent pratiquement rien. Par exemple, si vous avez 2 ou 4 variables locales en plus ou en moins dans une routine non récursive, le temps d'exécution ne changera quasiment pas. Dans le cas d'une routine récursive, le temps d'exécution ne changera pratiquement pas non plus sauf si la taille supplémentaire de la pile induit un *swap* de mémoire sur disque.



l'invariant d'itération correspondant. Sur le schéma mémoire de la vue 189 on peut constater le phénomène d'aliasing : les trois chaînes de la classe `STRING` sont toutes les trois référencées trois fois. Enfin, sur la vue 190, l'algorithme correspondant à l'invariant donné précédemment. L'intérêt de l'utilisation de l'aliasing dans cet exemple est double : d'une part on minimise le nombre d'objets de la classe `STRING` à créer (trois chaînes dans tous les cas) et d'autre part, il est possible d'utiliser l'opérateur élémentaire de comparaison `=` qui est beaucoup plus rapide que la fonction `is_equal`. Bien entendu, l'utilisation de l'opérateur `=` n'est rendu possible que par le fait que l'aliasing est maximal (il existe une et une seule chaîne "RoUgE" par exemple). De manière générale, l'expérience montre que le gain que l'on peut espérer en utilisant l'aliasing est souvent très important. Nous utilisons intensivement l'aliasing dans le compilateur<sup>2</sup> ainsi que dans la bibliothèque graphique `Vision`. Ceci étant, il faut également garder à l'esprit que l'utilisation de l'aliasing est une pratique très délicate et l'aliasing est souvent considéré, à juste titre, comme une pratique dangereuse qu'il faut éviter<sup>3</sup>. Grâce au mécanisme d'assertions, il est souvent possible de poser les garde fous permettant de contrôler raisonnablement la bonne utilisation de l'aliasing. Par exemple, pour revenir au codage de l'algorithme du drapeau tricolore, on peut noter à ce sujet l'utilisation d'une clause `check` dans la branche par défaut de l'étude de cas (vue 190).

### Aliasing (exemple avec l'exercice du drapeau)



```

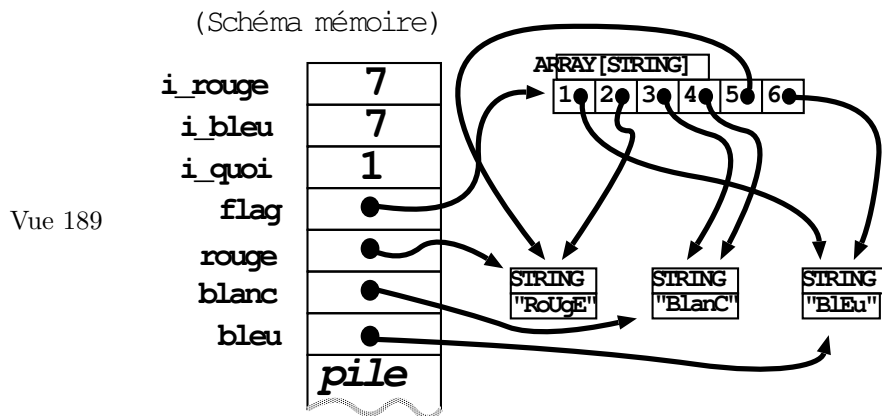
flag_sort is
  local bleu, blanc, rouge: STRING; flag: ARRAY[STRING]
  i_bleu, i_quoi, i_rouge: INTEGER; val_quoi: STRING
do
  from
    bleu := "BlEu";   blanc := "BlanC";   rouge := "RoUgE"
    flag := << bleu, rouge, blanc, blanc, rouge, bleu >>
    i_bleu := flag.lower - 1
    i_quoi := flag.upper ; i_rouge := i_quoi + 1
  variant
    i_quoi - i_bleu

```

Vue 188

<sup>2</sup>Pour une étude détaillée de l'utilisation de l'aliasing dans les outils de compilation, vous pouvez télécharger l'article [ZC00] [ZC01] (*Coping with aliasing in the GNU Eiffel Compiler implementation*) à partir du site web de SmartEiffel.

<sup>3</sup>Fils de mécanicien, mon avis personnel consiste à dire qu'il est plus difficile de mettre au point le moteur d'une ferarri que le moteur d'une deux chevaux ! Pour revenir à l'informatique, je ne connais pas beaucoup de logiciels performants qui n'utilisent pas massivement l'aliasing.



Vue 190

```

until  i_quoi = i_bleu
loop
  val_quoi := flag.item(i_quoi)
  if val_quoi = blanc then
    i_quoi := i_quoi - 1
  elseif val_quoi = bleu then
    i_bleu := i_bleu + 1
    flag.swap(i_quoi, i_bleu)
  else
    check val_quoi = rouge end
    i_rouge := i_rouge - 1
    flag.swap(i_quoi, i_rouge)
    i_quoi := i_quoi - 1
  end
end
end
end -- flag_sort

```

## 28.2 Éviter les fuites de mémoire

Une fuite de mémoire (*memory leak*) se produit lorsque le système qui s'exécute perd l'accès à une donnée préalablement allouée dans le tas. La vue 191 est un exemple manifeste d'énorme fuite mémoire : 999999 objets de la classe POINT sont alloués dans le tas sans qu'aucune référence ne soit gardée vers ces POINTS. Notons qu'il n'est pas correct de parler de fuite de mémoire lorsque qu'un processus est équipé d'un ramasse-miettes puisque ce système de récupération automatique de la mémoire garde une référence sur tous les objets alloués. Ceci étant, même avec un ramasse-miettes performant, il faut essayer d'éviter de créer inutilement des objets. Il faut être économe en ce qui concerne la création d'objets et certaines fuites de mémoire sont beaucoup moins évidentes que celle de la vue 191.

La vue 192 montre quelques fuites de mémoire pas forcément évidentes. Pour chaque fuite, une solution est proposée en bas de la même vue. La fuite #1 est liée au fait qu'une chaîne manifeste est créée lors de chaque évaluation de l'expression "Memory leak". La solution pour cette fuite #1 repose sur l'utilisation du mot clef **once** directement devant cette chaîne explicite (ce raccourci d'écriture est également autorisé devant les objets `U"foo"` de la classe UNICODE\_STRING). La fuite #2 est corrigée en évitant de convertir l'entier en une STRING avant de l'imprimer. La fuite #3 est corrigée en utilisant `append_in` de la classe INTEGER à la place de `append` de la classe STRING. Pour détecter les fuites de la catégorie #1, le compilateur dispose depuis peu d'une nouvelle option : `-manifest_string_trace` (se reporter à la documentation disponible sur le site web de SmartEiffel pour les détails). Pour les autres fuites de mémoire et de manière générale, pour mesurer la consommation mémoire par catégorie d'objets Eiffel alloués, le compilateur dispose également d'une option qui demande au ramasse-miettes de faire état des objets alloués (cf. option `-gc_info`).

**Exercice 56** Combien d'objets de la classe STRING sont alloués lors de l'exécution de l'instruction suivante :

```
print( "2" + (2).to_string + "=4" )
```

Donnez une suite d'instructions permettant d'afficher le même message en minimisant le nombre d'allocations de STRINGS.

## Une (grosse) fuite de mémoire

Vue 191

```

from
  i := 0
until
  i = 1000000
loop
  -- Memory leak here:
  create {POINT} variable.make(1,1)
  i := i + 1
end

```

## Fuites de mémoire moins évidentes

Vue 192

```

io.put_string( "Memory leak" )           -- Memory leak #1
io.put_string( an_integer.to_string )    -- Memory leak #2
a_string.append( an_integer.to_string )  -- Memory leak #3

```

### —— Solutions ——

```

io.put_string( once "Memory leak" )      -- Solution #1
io.put_integer( an_integer )             -- Solution #2
an_integer.append_in( a_string )         -- Solution #3

```

## 29 Appel de la super-méthode

Pour faire appel à la superméthode (i.e. la version héritée d'une méthode que l'on est en train de redéfinir), il existe une construction nommée `Precursor` en Eiffel. Le mot clef `Precursor` n'est utilisable qu'à l'intérieur des méthodes pour lesquelles il existe une indication de redéfinition (clause `redefine`). L'appel de `Precursor` dans une méthode donnée à la même signature que la méthode qui l'englobe. Ceci permet de faire l'appel de la version précédemment définie de la méthode en question. Sur l'exemple de la vue 193 le calcul de la nouvelle définition de `prix.ttc` dans `ARTICLE_DEUX_GRAND_LUXE` consiste à multiplier par deux le résultat du calcul de `prix.ttc` de la classe `ARTICLE_DE_LUXE`. L'exemple de la vue 194 montre comment il est possible d'utiliser ce mécanisme dans le cas d'une procédure avec deux arguments (i.e. l'appel de `Precursor` correspond à un appel de procédure ayant la même signature).

### Precursor : appel de la superméthode

```

class ARTICLE_DEUX_GRAND_LUXE
inherit
  ARTICLE_DE_LUXE
  redefine prix_ttc
end

Vue 193 -- ...
-- ...

prix_ttc: REAL is
  -- De grand luxe, donc DEUX fois plus cher !
  do
    Result := 2 * Precursor
  end -- prix_ttc
end -- class ARTICLE_DEUX_GRAND_LUXE

```

```

class F00
inherit
  ARTICLE
  redefine bar
end

-- ...
-- ...

bar (arg1: CHARACTER; arg2: REAL) is
do
  -- Code que l'on souhaite ajouter avant l'appel à la superméthode.
  Precursor('A', 1.5) -- ← Appel de la superméthode.
  -- Code que l'on souhaite ajouter après l'appel à la superméthode.
end -- bar
end -- class ARTICLE_DE_GRAND_LUXE

```

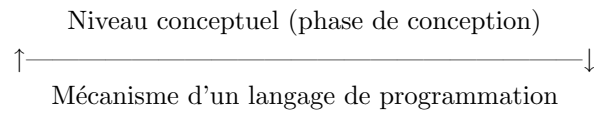
Vue 194

### 30 Conclusion - Remarques

Il ne faut pas confondre les relations conceptuelles qui existent entre les types (spécialisation, abstraction, représentation, union de types) avec une relation d'héritage entre deux classes. Exemple si B est une classe qui hérite de A, on ne sait pas grand chose sur la relation qui existe entre le type A et le type B. Ces deux classes peuvent avoir des interfaces complètement différentes !

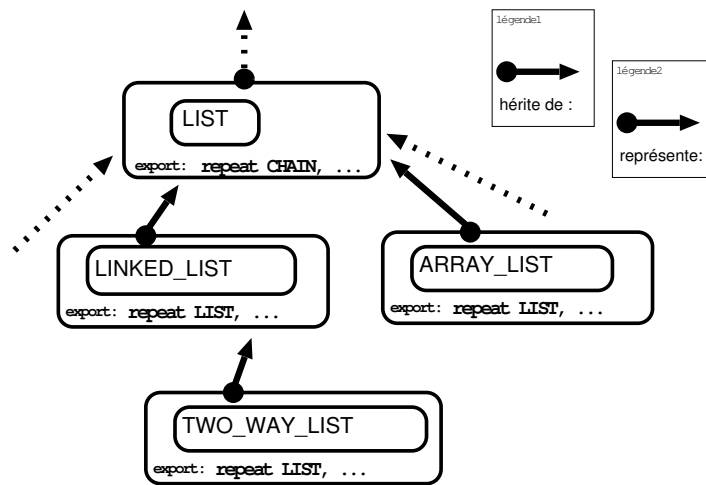
**SPÉCIALISATION**  
**ABSTRACTION/REPRÉSENTATION**  
**UNION DE TYPES**

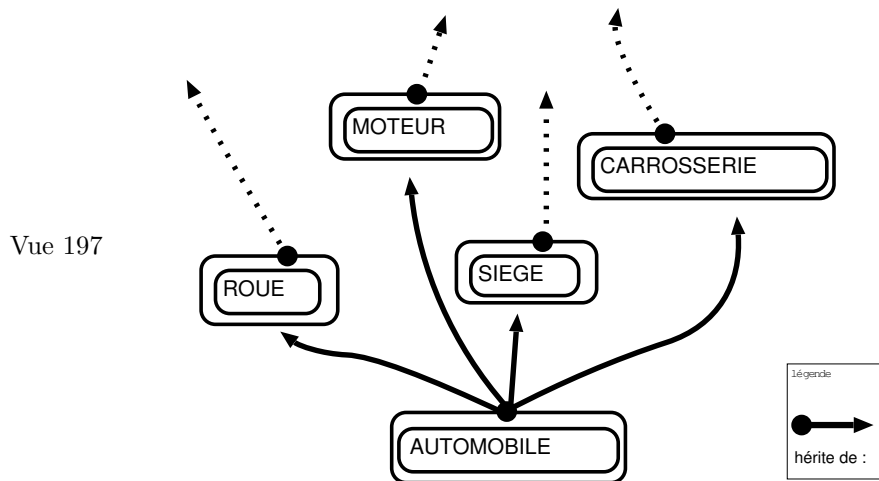
Vue 195



**HÉRITAGE**

Vue 196





### composition

Vue 198

```

class AUTOMOBILE
-- N'HÉRITE DE RIEN.
creation
  make
feature

  roueAvD, roueAvG, roueArD, roueArG : ROUE
  unMoteur : MOTEUR
  caisse : CARROSSERIE
  interieur : SIEGE
  ...
end -- class AUTOMOBILE
  
```



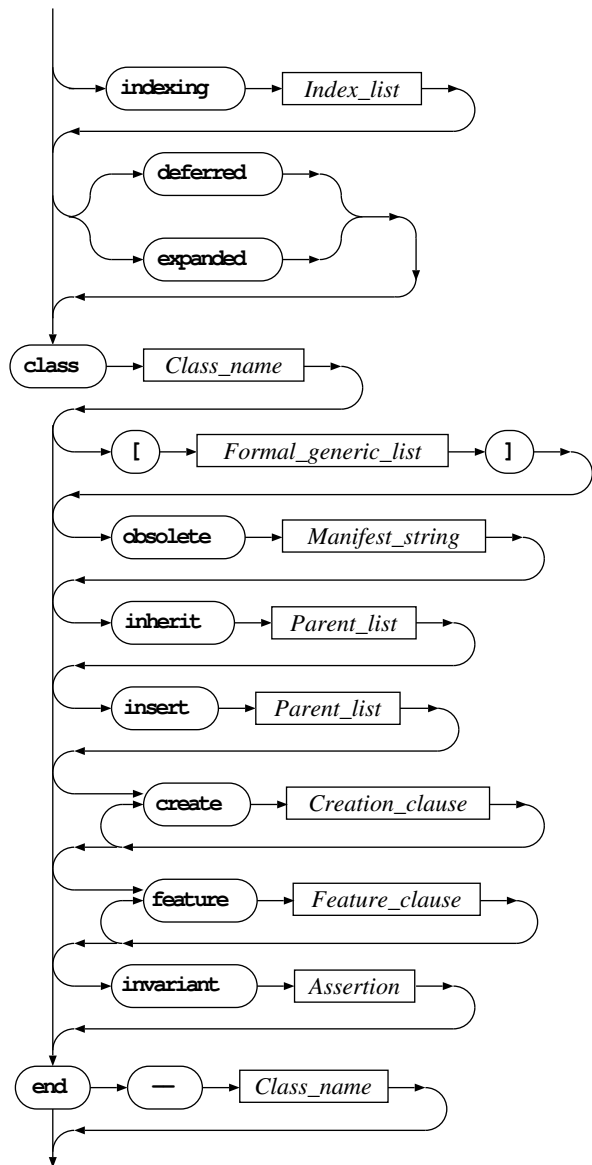
```
class A
  inherit B
  ...
end -- class A
```

Vue 199

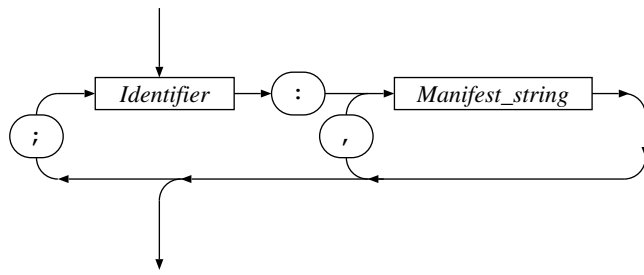
**S'agit il vraiment d'une sorte de B ?**

### 31 Diagrammes syntaxiques

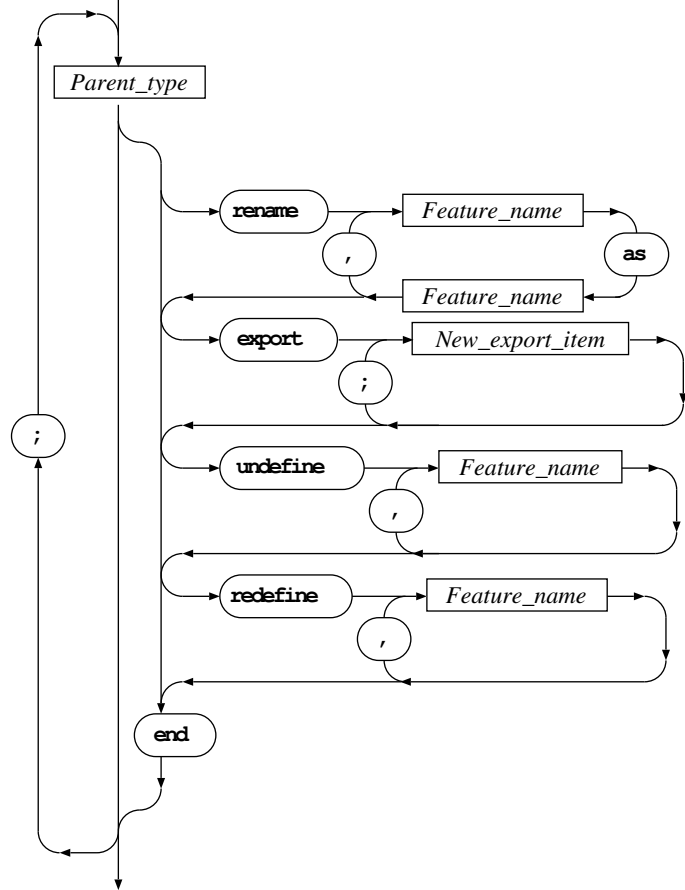
*Class\_declaration*



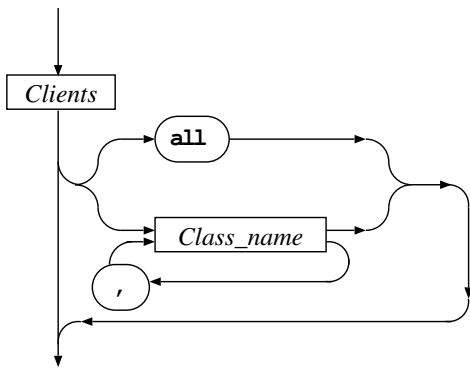
*Index\_list*



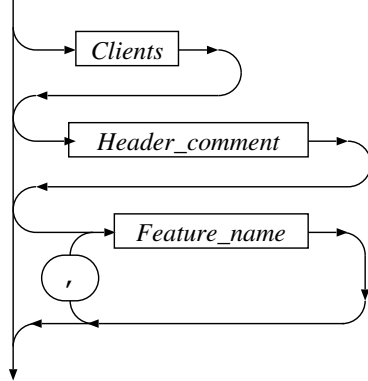
*Parent\_list*



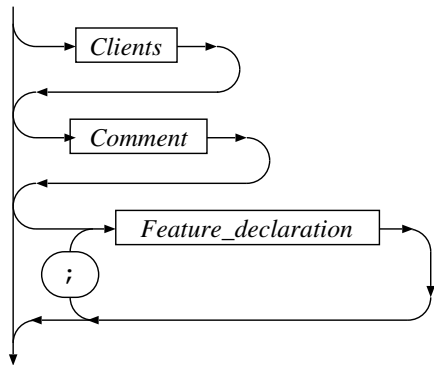
*New\_export\_item*



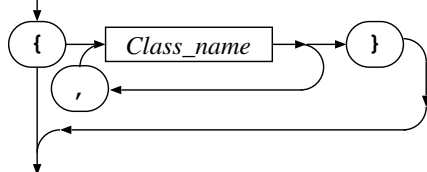
*Creation\_clause*



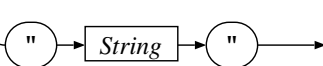
*Feature\_clause*



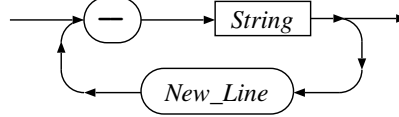
*Clients*



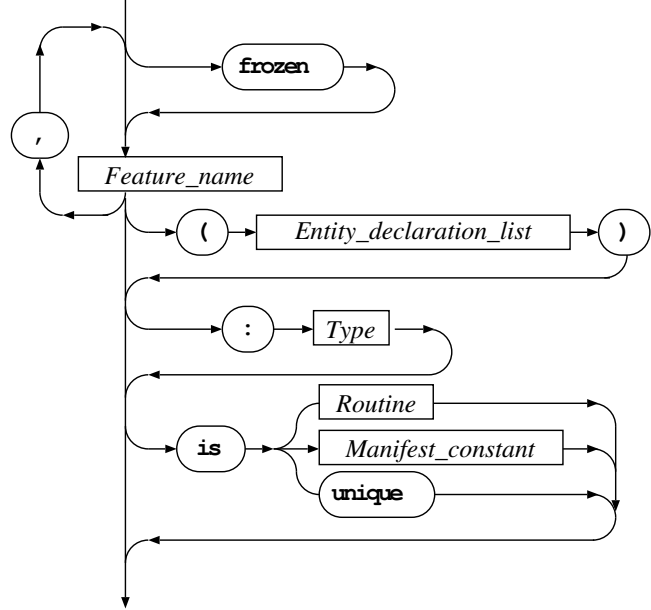
*Manifest\_string*



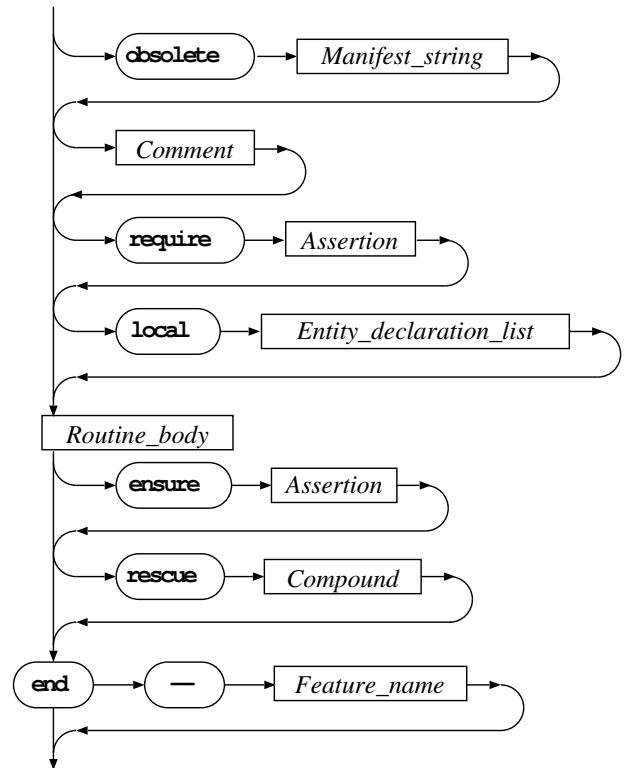
*Comment*

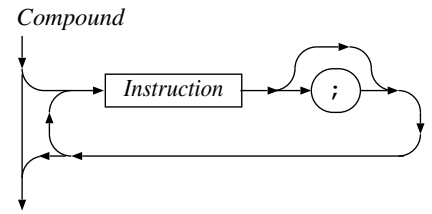
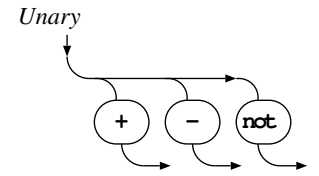
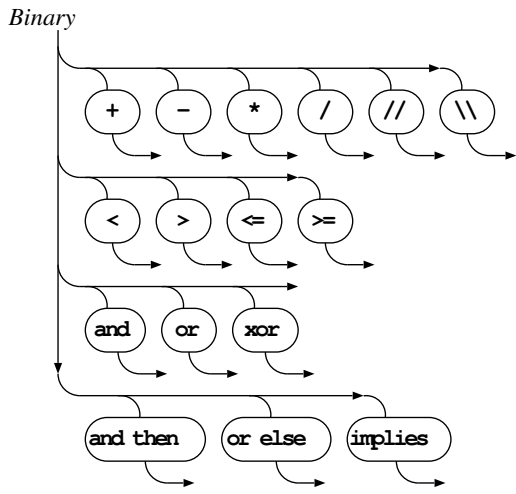
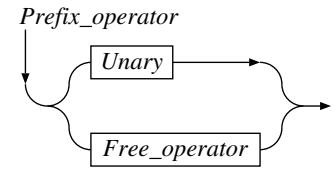
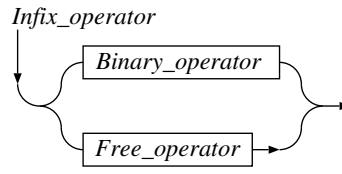
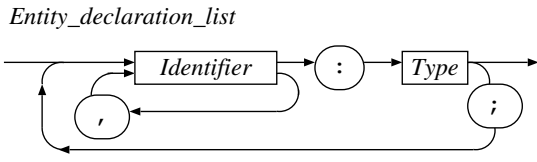


*Feature\_declaration*

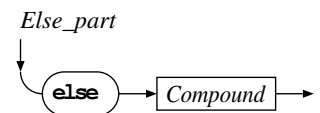
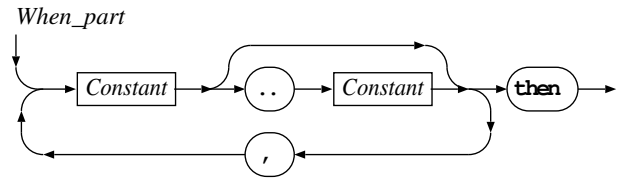
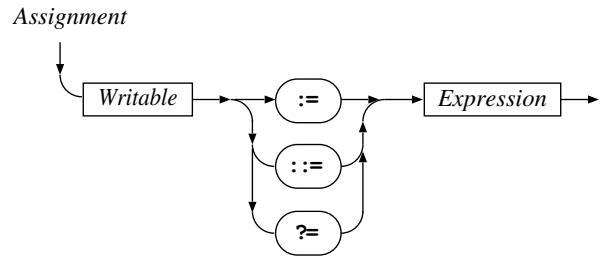
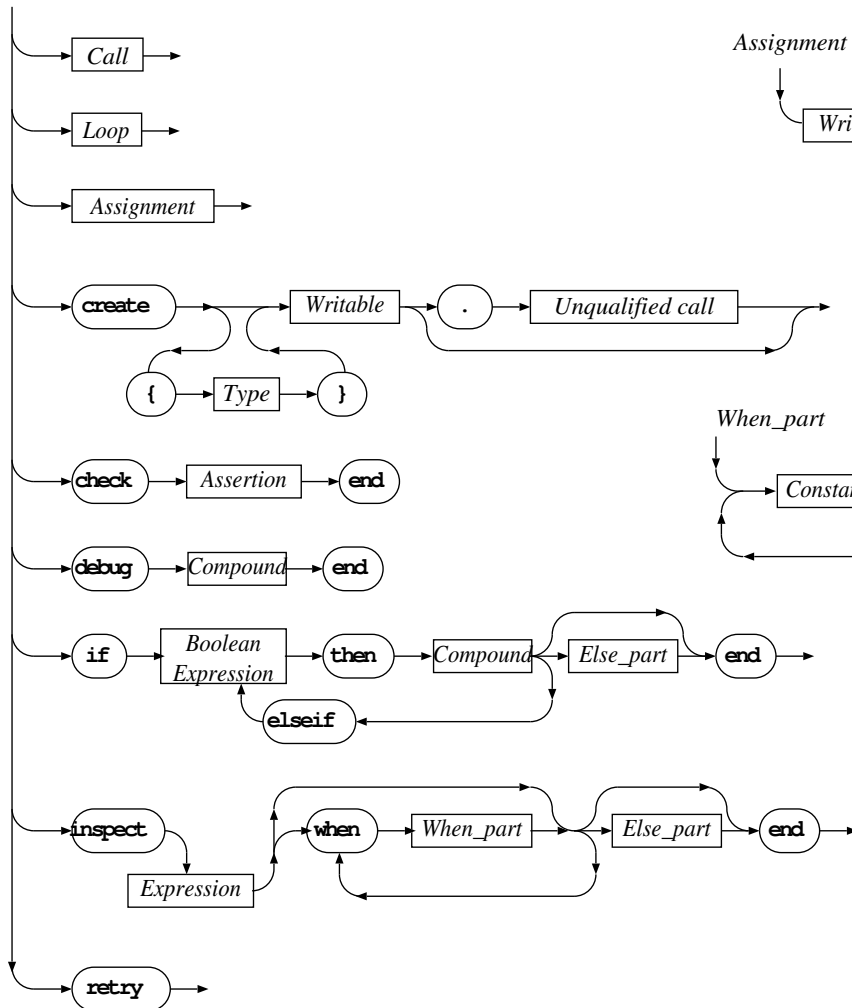


*Routine*

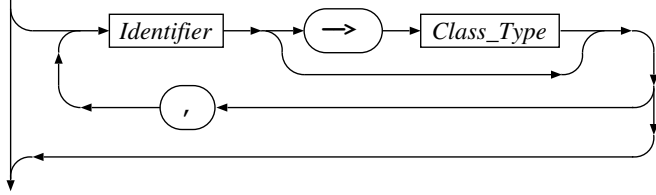




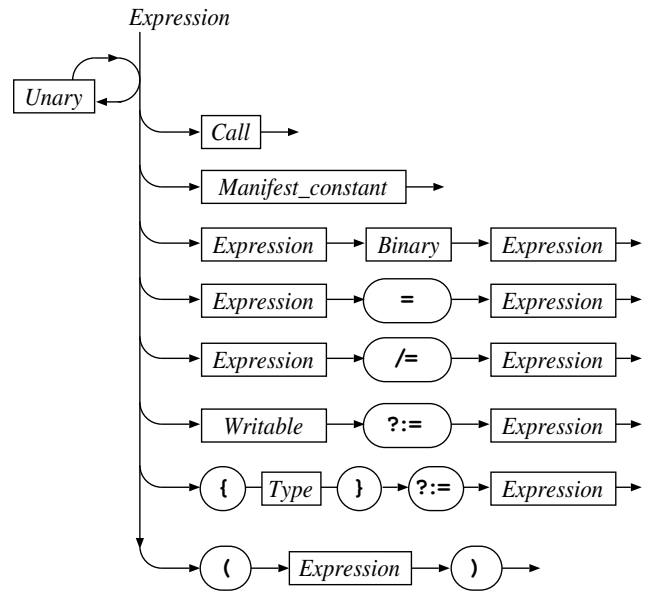
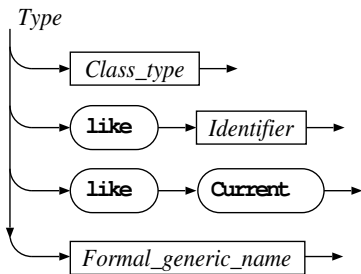
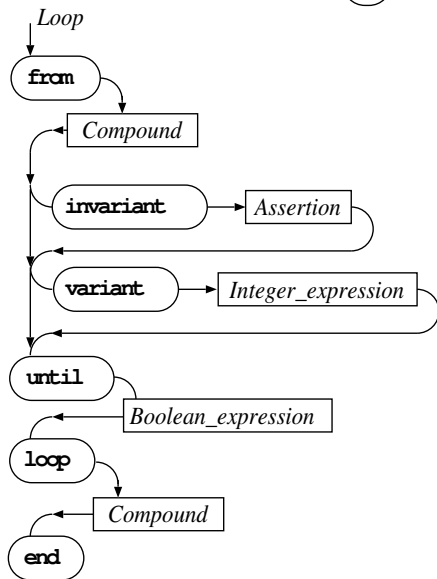
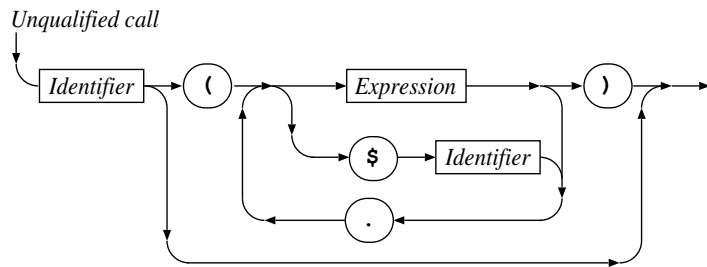
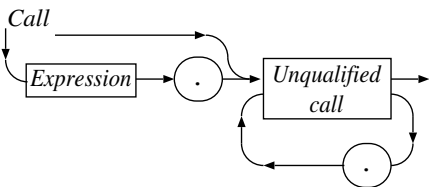
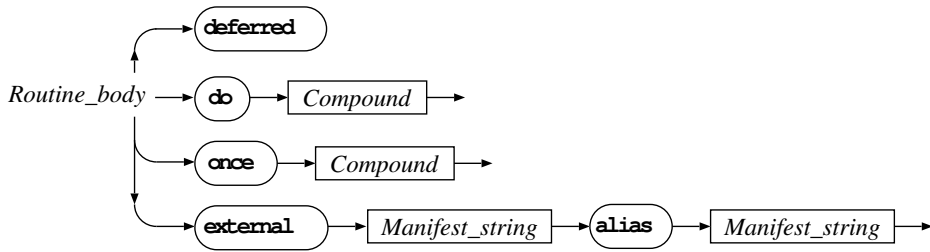
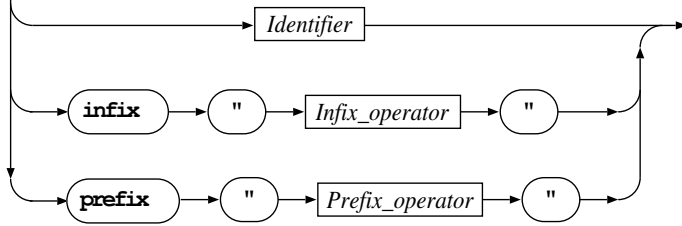
*Instruction*



Formal\_generic\_list



Feature\_name



## Références

- [CCZ97] S. Collin, D. Colnet, and O. Zendra. Type Inference for Late Binding. The SmallEiffel Compiler. In *Joint Modular Languages Conference, JMLC'97*, volume 1204 of *Lecture Notes in Computer Sciences*, pages 67–81, Lintz, Austria, 1997. Springer Verlag. LORIA 98-R-056. Disponible sur <http://www.loria.fr/~colnet/publis/index-us.html>.
- [CCZ98] D. Colnet, P. Coucaud, and O. Zendra. Compiler Support to Customize the Mark and Sweep Algorithm. In *ACM SIGPLAN International Symposium on Memory Management (ISMM'98)*, pages 154–165, Vancouver, BC, Canada, October 1998. LORIA 98-R-233. Disponible sur <http://www.loria.fr/~colnet/publis/index-us.html>.
- [CZ99] D. Colnet and O. Zendra. Optimizations of Eiffel programs : SmallEiffel, The GNU Eiffel Compiler. In *29th conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'99)*, Nancy, France, pages 341–350. IEEE Computer Society PR00275, June 1999. LORIA 99-R-061. Disponible sur <http://www.loria.fr/~colnet/publis/index-us.html>.
- [JTM99] J. M. Jézéquel, M. Train, and C. Mingins. *Design Patterns and Contracts*. Addison-Wesley, 1999. ISBN 0-201-30959-9.
- [MNC<sup>+</sup>89] G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Tombre. *Les langages à objets*. InterEditions, Paris, 1989. LORIA 89-R-126.
- [ZC99] O. Zendra and D. Colnet. Adding external iterators to an existing Eiffel class library. In *32th conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'99)*, Melbourne, Australia, pages 188–199. IEEE Computer Society PR00462, November 1999. LORIA 99-R-318. Disponible sur <http://www.loria.fr/~colnet/publis/index-us.html>.
- [ZC00] O. Zendra and D. Colnet. Vers un usage plus sûr de l'aliasing avec Eiffel. In *5ème Colloque Langages et Modèles à Objets (LMO'2000)*, Mont St-Hilaire, Québec, pages 183–194. Hermes, Janvier 2000. LORIA A00-R-022. Disponible sur <http://www.loria.fr/~colnet/publis/index-fr.html>.
- [ZC01] O. Zendra and D. Colnet. Coping with aliasing in the GNU Eiffel Compiler implementation. *Software Praticce and Experience (SP&E)*, 31(6) :601–613, 2001. J. Wiley & Sons. Disponible sur <http://SmartEiffel.loria.fr>.
- [ZCC97] O. Zendra, D. Colnet, and S. Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, volume 32, number 10 of *SIGPLAN Notices*, pages 125–141. ACM Press, October 1997. LORIA 97-R-140. Disponible sur <http://SmartEiffel.loria.fr>.

# Index

=, 35

abstraction, 166  
affectation, 19  
*aliasing*, 37, 47, 160  
ANY, 33, 134, 137  
argument effectif, 46  
argument formel, 46  
ARRAY, 47, 62  
ARTICLE, 113  
ARTICLE\_DE\_LUXE, 125  
assertion, 9, 73, 74, 126  
attribut, 16–18, 26  
attribut (écriture), 33  
AVL\_DICTIONARY, 63  
AVL\_SET, 62  
  
BIT\_N, 23  
-boost, 9, 26  
*byte-code*, 7  
  
chaîne de caractères, voir `STRING`  
CHARACTER, 23  
check, 74, 161  
cible, 25  
class\_check, 6  
classe, 16–18  
classe abstraite, 145  
classe racine, 7  
client, 31, 33  
client (relation), 33, 85  
clone, 156, voir `twinn`  
cohérence globale, 142  
cohérence locale, 142  
COLLECTION, 156  
COMPARABLE, 145, 147  
comparaison d'objets, 35  
compile, 3, 6, 9, 40  
compile\_to\_c, 6  
compile\_to\_jvm, 6  
concordance de types, 128  
conflit d'héritage, 154  
constructeur, 28  
contrat, 73  
copier un objet, 41  
copy, 48, 63, 156  
couplage, 33  
création, voir `instanciation`  
create, 101  
creation, 28, 145

Current, 28, 46, 85, 125, 128

débogueur, 19, 26, 37, 83, voir `sedb`  
dérivation, 111  
*dangling pointer*, 40  
debug, 74  
deferred, 145  
*design pattern*, 10  
diagrammes syntaxiques, 170  
DICTIONARY, 63  
DOUBLE, 23  
drapeau tricolore, 160  
duplication d'objets, 35  
  
ensemble, voir `SET`  
ensure, 73, 74  
entrées sorties, 71  
equal, 156, voir `is_equal`  
expanded, 19, 23, 46  
export, 142  
exportation, 33, 125  
expression, 101  
  
FAQ, 4, 10  
FAST\_ARRAY, 62  
feature, 33, 122  
finder, 4, 6  
*Flyweight*, 95  
fonction, 26  
fournisseur, 31  
frozen, 137, voir `geler`  
fuite de mémoire, 40, 163  
  
généricite, 108, 111, 147, 148  
généricite contrainte, 147  
généricite non contrainte, 108  
-gc\_info, 41  
-gc\_info, 163  
geler, 137, voir `frozen`  
GENERAL, 134, 137, 156  
  
héritage, 112, 122, 126, 166  
héritage des assertions, 126  
héritage multiple, 154  
HASHED\_DICTIONARY, 63  
HASHED\_SET, 62  
*heap*, voir `tas`  
HELLO\_WORLD, 3  
-help, 6  
historique des langages à objets, 12  
<http://SmartEiffel.loria.fr>, 3

- `inherit`, 125, 154, voir héritage
- initialisation des attributs, 23
- initialisation des variables, 23
- instance, 16–18
- instanciation, 16, 18, 28, 46, 74, 111, 132
- instanciation (nouvelle notation), 101
- instruction, 101
- INTEGER, 23
- interface, 33, 166
- invariant, 73, 74, 122
- invariant d’itération, 74
- io, 71
- `is_equal`, 35, 137, 147, 156, 157
- itération, 74
- iterator*, 10
  
- Java, 7, 23, 33
  
- langage C, 7
- liaison dynamique, 25, 128, 132, 134, 151, 154
- like, 128
- like Current, 128
- LINKED\_LIST, 62
- liste, voir LINKED\_LIST
- liste SmartEiffel, 4
- local, 18
- loop, 74
  
- méthode, 26, 85, 121, 123, 125, 126, 132, 147
- mailing liste*, 4
- `-manifest_string_trace`, 163
- masquage, 132
- memory leak*, 163
- MEMORY, 40
- message, 25
- modificateur, 73, 93
- module, 33
- mouchard, 74
  
- NATIVE\_ARRAY, 62
- `-no_gc`, 40
- NONE, 33, 137
  
- observateur, 93
- old, 73
- once, 93
- opérateur `?=`, 137
- opérateur `=`, 35
- opérateurs (priorité), 97
- options de compile, 6
- options de `compile_to_c`, 6
- organisation mémoire, 24
  
- paramètre générique, 111
- partie privée, 33
- partie publique, 33
- passage d’arguments, 46
- patron de conception, 10
- pgcd, 74
- pile, 24, 160
- PILE\_FIXE, 73
- PILE\_MOCHE, 139
- PLATFORM, 137
- POINT, 18, 19, 23, 31, 33, 37, 85
- POINTER, 23
- pointeur, 23
- pointeur invalide, 40
- POLYGONE, 128
- polymorphisme de variable, 128
- post-condition, 73, 74, 137
- pré-condition, 73, 74
- Precursor, 165
- pretty, 6
- primitive, 26, 33, 125
- principe de référence uniforme, 33
- priorité des opérateurs, 97
- procédure, 26
- programmation par contrat, 73
- programme principal, 7
  
- référence, 23, 37
- ramasse-miettes, 19, 24, 37, 40, 41, 163
- receveur, 25, 46
- RECTANGLE, 128
- redéfinition, 126, 137, 156
- redefine, 125, 152
- rename, 123
- représentation, 166
- require, 73, 74
- RING\_ARRAY, 62
- routine, 16, 17, 26
  
- sélecteur, 25
- schéma mémoire, 19, 24, 37, 41, 42
- `-sedb`, 10, 19, 83
- sedb, 83
- SET, 62
- short, 6
- SORTED\_LIST, 147
- sous-classe, 116, 121, 122, 126, 137, 145
- spécialisation, 115, 166
- stack*, voir pile
- STD\_ERROR, 71
- std\_error, 71
- STD\_INPUT, 71
- std\_input, 71
- STD\_OUTPUT, 71



- `std_output`, 71
- `STRING`, 47
- superclasse, 116, 121, 122, 154
- superméthode, 123, 137, 165
  
- table, 63, voir `DICTIONARY`
- tableaux, 62, voir `ARRAY`, `FAST_ARRAY`
- target*, 25, voir cible
- tas, 24, 160
- `TRIANGLE`, 31, 33
- `sctriangle`, 37
- tuple*, 10
- `twin`, 41
- `twin`, voir `copy`
- `TWO_WAY_LINKED_LIST`, 62
- type dynamique, 128, 132, 138
- type `expanded`, 19, 23, 46
- type statique, 128, 132
  
- `UNICODE_STRING`, 163
- union de types, 166
  
- variable d'instance, 18, 26
- variable globale, 18
- variable locale, 18
- `variant`, 74
- vecteur, voir tableaux
- `-verbose`, 5
- `-version`, 9
- `VETEMENT`, 115
- visibilité, 122, voir exportation
- `Void`, 19, 25, 28, 31, 137
- `Void target`, 25