Chapitre 5

Références, visibilité des variables

1 Un objet est identifié par une *référence*

1.1 Notion de référence

En langage objet, les variables d'un type objet (c'est à dire d'un type non primitif) contiennent comme valeurs des *références* à des objets.

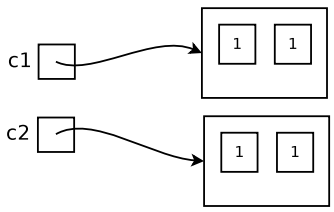
La référence à un objet est un identifiant indiquant de manière non ambiguë de quel objet on parle. On peut l'interpréter comme la désignation de l'endroit où trouver l'objet en mémoire, c'est à dire comme son adresse.

L'opérateur creer (new en Java) construit un objet et retourne une référence à cet objet.

Exemple.

```
NombreComplexe c1 ;
NombreComplexe c2 ;
c1 ← creer NombreComplexe() ;
c2 ← creer NombreComplexe() ;
```

déclarent deux variables de type NombreComplexe, et instancient successivement deux objets de type NombreComplexe. La variable c1 prend pour valeur une référence au premier de ces deux objets, tandis que c2 prend pour valeur une référence au deuxième objet.



Une variable d'un type objet a la valeur nul (null en Java) quand elle ne fait référence à aucun objet instancié. C'est le cas quand elle a été déclarée mais qu'aucune référence ne lui a encore été affectée par creer.

La notion de référence s'applique uniquement aux variables de type objet, pas aux variables de type primitif.

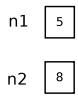
Abus de langage. Par abus de langage, on dira qu'une variable d'un type objet *est* une référence à un objet, au lieu de dire qu'elle *contient* une référence à l'objet.

Exemple. Dans l'exemple précédent, on dira abusivement que $c1$ et $c2$ sont des références à des nombres complexes.

1.2 Affectation d'une référence à une autre

Considérons deux variables entières (et donc de type primitif) n1 et n2, et les instructions

```
n1 ← 5 ;
n1 ← 8 ;
```



L'instruction

```
n1 ← n2 ;
```

a ensuite pour effet pour effet que n1 prend la valeur de n2.

Mais ensuite, si on change la valeur de n2, par exemple par l'instruction

```
n2 ← n2+1 ;
```

alors n1 et n2 ne contiennent plus la même valeur.

L'affectation de n2 à n1 n'a pas fait coïncider n2 et n1.

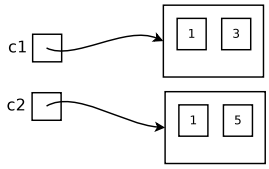
Les choses seraient différentes si n1 et n2 n'étaient pas de type primitif, et qu'elles étaient (ou plutôt qu'elles contenaient) deux *références* à des objets. L'affectation d'une référence à une autre fait coïncider les deux objets.

Considérons les deux nombres complexes c1 et c2 de tout à l'heure. On peut affecter à c1 une partie imaginaire de 3 :

c1.plmag ← 3 ;

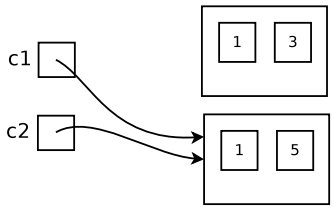
puis à c2 une partie imaginaire de 5 :

c2.plmag ← 5 ;



Que se passe-t-il maintenant si je réalise l'affectation

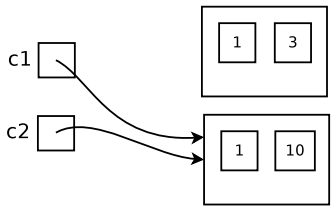
c1 ← c2 ; ?



Comme on peut s'y attendre, la partie imaginaire de c1 passe de 3 à 5. Mais si j'affecte à c1 une nouvelle partie imaginaire, par exemple

c1.plmag ← 10 ;

alors la partie imaginaire de c2 devient elle aussi égale à 10 !



L'affectation

c1 ← c2 ;

fait coïncider c1 et c2. En effet, c1 ← c2 ; signifie à proprement parler : « la référence contenue par $c1$ devient égale à celle contenue par $c2$ ». Autrement dit, c1 et c2 désignent le *même* objet en mémoire. L'objet anciennement désigné par la référence contenue dans c1 est perdu, et l'objet dont la référence est contenue par $c2$ a maintenant sa référence contenue aussi par $c1$. C'est pourquoi toute opération sur c1 affecte également c2 et réciproquement.

Ramasse-miettes (ou *Garbage collector* en anglais) En java, lorsqu'un objet n'est plus référencé par aucune variable (voir par exemple l'objet anciennement désigné par c1 de tout à l'heure), l'espace mémoire qu'il occupait est automatiquement récupéré. Ce mécanisme est connu sous le nom de *ramasse-miettes*. Ce mécanisme n'est pas celui de C++, où le programmeur doit explicitement libérer l'espace mémoire des objets qu'il n'utilise plus.

1.3 Passage par valeur ou passage par adresse ?

On a déjà dit que Java ne proposait que le passage par valeur. De manière concrète, cela signifie que

les paramètres d'une méthode sont des variables locales à la méthode, dans lesquelles les valeurs des variables passées en paramètre sont recopiées.

Mais si un paramètre est d'un type objet, cela signifie que la *valeur* qu'il reçoit (et recopie localement) est une référence à un objet. En conséquence, toute modification apportée à l'objet référencé lui-même est persistante à la sortie de la fonction ou de l'action. Ce mécanisme fait parfois dire à certains (à tort !) qu'il y'a du passage par adresse en Java. Il n'y a bien que du passage par valeur, mais les valeurs en question peuvent être des références à des objets. La valeur de cette référence ne peut pas être modifiée mais l'objet référencé si !

1.4 Référence à soi-même : le mot-clé **moi** (ou **this**)

La syntaxe objet impose que l'on désigne un attribut att d'un objet obj par la syntaxe obj.att, et une méthode meth() d'un objet obj par la syntaxe obj.meth(). À l'intérieur même d'une classe, cette syntaxe n'est cependant pas respectée : on désigne par exemple l'attribut pReelle de la classe NombreComplexe directement par pReelle. Il faut dire qu'à *l'intérieur* de la classe NombreComplexe, on ne sait pas sous quel nom on sera désigné à *l'extérieur* !

Un nom spécial a été prévu pour faire référence à l'objet tel qu'il sera vu de l'extérieur : il s'agit du mot clé moi, qui signifie « référence à moi-même ». En Java, ce mot clé est this.

Dans la classe NombreComplexe par exemple, le code de la méthode module(), au lieu de s'écrire

```
retourner racine(pReelle*pReelle+pImag*pImag) ;
```

aurait pu s'écrire

```
retourner racine(moi.pReelle*moi.pReelle + moi.pImag*moi.pImag) ;
```

Utiliser le mot-clé moi peut être utile par exemple pour donner à un paramètre le même nom que l'attribut pour lequel il est censé contenir une valeur. C'est d'ailleurs l'écriture recommandée en objet. Ainsi, une bonne écriture du constructeur à deux paramètres de la classe NombreComplexe est :

```
Constructeur(réel pReelle, réel pImag)
    moi.pReelle ← pReelle ;
    moi.pImag ← pImag ;
fin
```

ce qui donne en Java :

```
NombreComplexe(double pReelle, double pImag) {
    this.pReelle = pReelle ;
    this.pImag = pImag ;
}
```

Ici, les paramètres pReelle et pImag *masquent* les attributs pReelle et pImag, et il faut donc expliciter ces derniers.

2 Portée des variables en Java

2.1 Notion de bloc

Un bloc en Java est une suite d'instructions délimité, explicitement ou implicitement, par une paire d'accolades ouvrante et fermante. Le code d'une classe Java est ainsi structuré en un ensemble de blocs, dont certains sont imbriqués les uns dans les autres.

Exemple. On considère la classe suivante

```
public class Bidule {
    int i, j, k, z;                                          bloc 1
    public void truc( int z ) {
        int j, r;                                            bloc 2
        if (z < 0)
            r = -z;                                          bloc 4 (implicite)
        else
            r = z;                                           bloc 5 (implicite)
    }

    public void machin( ) {
        k = r;                                               bloc 3
    }
}
```

Il y'a trois blocs explicites :
– **bloc 1 :** celui qui définit le code de la classe Bidule(),
– **bloc 2 :** celui qui définit le code de la méthode truc(),
– **bloc 3 :** celui qui définit le code de la méthode machin(),
Le bloc 2 (de truc()) est englobé dans le bloc 1 (de Bidule()). Le bloc 3 (de machin()) est lui aussi englobé dans le bloc 1.

N.B. Les accolades ouvrantes et fermantes sont parfois implicites, comme dans l'exemple

```
if (x1 ≤ x2)
        System.out.println("x1 est plus petit que x2");
else
        System.out.println("x1 est plus grand que x2");
```

qui doit se lire comme

```
if (x1 ≤ x2) {
        System.out.println("x1 est plus petit que x2");   (un bloc)
}
else {
        System.out.println("x1 est plus grand que x2");   (un autre bloc)
}
```

En effet, comme les branches de ce *if . . . then . . . else* se limitent à une seule instruction, la syntaxe Java autorise d'omettre les accolades les délimitant.

Il y'a donc 5 blocs en tout dans la classe Bidule : **bloc 1**, **bloc 2** et **bloc 3** qui sont explicites, mais aussi **bloc 4** et **bloc 5** qui sont implicites.

2.2 Portée des variables

Les variables sont visibles et ne sont visibles qu'à l'intérieur du *bloc* (et donc des sous-blocs) dans lequel elles sont déclarées.

N.B. La déclaration d'un paramètre d'une méthode est considérée comme appartenant au bloc que définit le corps de la méthode.

Ainsi, dans l'exemple précédent (classe Bidule), la variable z paramètre de la méthode truc(int z) appartient au bloc 2, définissant le code de la méthode truc().

Dans la méthode truc(), les instructions r = z ; et r = -z : sont autorisées car z et r sont deux variables visibles dans tout le bloc 2, et donc y compris aussi dans ses sous-blocs 4 et 5.

La méthode machin() ne possède ni paramètre ni variable locale. La variable k y est pourtant visible car elle est connue dans tout le bloc 1, celui de la classe Bidule, qui contient le bloc 3.

Mais l'instruction `k = r;` de la méthode `machin()` provoque une erreur de compilation car r n'est pas visible dans le bloc 3.

Mécanisme de recherche. Quand une variable est utilisée, sa déclaration est recherchée dans le bloc où elle apparaît. Si elle ne s'y trouve pas, la recherche est étendue au bloc englobant, puis si besoin au bloc qui englobe le bloc englobant, etc. Si la recherche échoue, alors la variable n'est pas visible dans le bloc où elle est utilisée.

Conséquence. En cas de conflit entre deux noms de variables, c'est toujours la variable « la plus locale » qui est considérée comme étant celle que l'on désigne.

Par exemple, la variable j du code de la méthode `truc()` est celle déclarée localement à la méthode. Ce n'est pas l'attribut j. De même pour la variable z de la méthode `truc()`. Elle désigne le *paramètre* z, et non pas l'attribut z.

Rappel. L'attribut d'une classe peut toujours être désigné sans ambiguïté en le préfixant par `this`.