

Remarques : dans les exemples donnés ici,

- --> est le *prompt* (invite) du shell utilisé ;
- on utilise `System.out` plutôt que la classe `Deug` pour réaliser des impressions. Cela a le même effet et est plus portable.

Rappels Java et illustrations des différences entre variables de types primitifs et variables de types références

```
--> cat Adresse1.java
// contenu du fichier Adresse1.java
// la compilation par la commande javac produit Adresse1.class
class Adresse1 {
    public static void main(String[] args) {
        int n = 10;    // type primitif
        int[] t1;     // type reference : tableau d'entiers
        int[][] t2;   // type reference : tableau de tableaux d'entiers
        t1 = new int[3];
        t2 = new int[4][]; // : tableau d'entiers
        System.out.println(n);
        System.out.println(t1);
        System.out.println(t2);
        for(int i = 0; i < t2.length; i++)
            System.out.println(t2[i] + " ");
    }
}
--> javac Adresse1.java
--> java Adresse1
10
[I@82f0db      <== t1 est un tableau d'entiers : adresse 82f0db
[[I@92d342    <== t2 est un tableau de tableau d'entiers : adresse 92d342
null          <== t2[0] a la valeur null
null          <== t2[1] a la valeur null
null          <== t2[2] a la valeur null
null          <== t2[3] a la valeur null et t2.length a la valeur 4
-->
```

Les variables de type tableau sont des variables de type référence. Il en est de même, par exemple, des variables de type `String`.

Les adresses sont données sous forme hexadécimale (par exemple `82f0db`). Une adresse est précédée du type de l'objet référencé (par exemple `[I` pour un tableau de valeurs de type `int` ou `[[I` pour un tableau de tableau de valeurs de type `int`).

Les valeurs de variables de ce type sont des **références**.

Un autre exemple : une classe `Couple` définie de la manière suivante :

```
--> cat Couple.java
class Couple {
    int a, b;
    String nom;
}
```

et un programme l'utilisant :

```
--> cat UseCouple.java
// la définition de Couple pourrait être dans ce fichier : la compilation
// de ce fichier créerait alors deux fichiers .class (Couple et UseCouple)
class UseCouple {
    public static void main(String[] args) {
        Couple c;
        c = new Couple();
        System.out.println(c);
        System.out.println(c.a); // accès au membre a
        System.out.println(c.b); // accès au membre b
        System.out.println(c.nom); // accès au membre nom
    }
}
--> java UseCouple
Couple@8fe7c3 <== c fait référence à un objet de type Couple
0 <== le champ a est initialisé à 0
0 <== le champ b est initialisé à 0
null <== le champ nom (type String) est une référence initialisée à null
-->
```

La valeur affichée pour la variable *c* indique qu'il s'agit d'une référence sur un objet de type `Couple` et que l'objet référencé est à l'adresse `8fe7c3`.

Notion d'adresse

Dans certains langages, par exemple Pascal, C ou C++, les choses sont un peu différentes. On peut manipuler effectivement les adresses et en particulier obtenir l'adresse d'un objet en mémoire.

Ainsi en C :

```
--> cat couple.c
typedef struct {int a, b; } couple;
main() {
    couple c; // déclare et alloue une variable de type couple
    c.a = 3; // affecte 3 au champ a de la structure
    c.b = 5; // affecte 5 au champ b de la structure
    printf("%p\n", &c); // imprime l'adresse de la variable c
}
--> gcc couple.c -o couple <== compilation
--> ./couple <== demande d'exécution
0xbffffd20 <== adresse en mémoire (sous forme hexa) de la variable c
-->
```

On y a défini un type `couple`. Puis après avoir défini une variable de type `couple`, on a affiché l'adresse du couple.

De fait, la variable *c* désigne directement la valeur du couple et non sa référence comme c'est le cas en java pour un objet de même nature. Par contre il est effectivement possible d'en obtenir l'adresse au travers de l'opérateur `&`.

En C et C++, il est possible de définir des **variables de type pointeur**.

```
int *p; // définit une variable de type pointeur sur entier.
couple *pc; // variable de type pointeur sur couple
```

et après de telles déclarations, si *i* et *c* sont respectivement des variables de type `int` et `couple`, on peut écrire :

```

int i = 4;
couple c;
....
p = &i; // la variable p contient l'adresse de i
pc = &c; // la variable pc contient l'adresse de c

```

Remarque : en Java il n'y pas cette possibilité d'obtenir une référence sur une variable de type `int`. Pour pallier cette impossibilité, un type non primitif nommé `Integer` a été créé (ce type est un *wrapper*, enveloppe d'entier).

Une version simplifiée de ce type est :

```

class Integer {
    int valeur; // la valeur du nombre;
    // définition d'un constructeur d'Integer
    Integer(int n){valeur = n;}
}

```

Un exemple d'utilisation en est :

```

Integer refi;
refi = new Integer(5);

```

La variable *refi* fait ainsi référence à un objet de type `Integer`. Cette séquence alloue l'espace pour une référence, puis l'espace pour un objet de type `Integer` alors qu'une définition de la forme

```
int n = 5;
```

n'alloue que l'espace nécessaire au stockage d'un entier (4 octets).

Déréférencement et indirection

- **indirection** : il correspond à la possibilité d'accéder à l'adresse d'une entité (objet) désignée par une variable ;
- **déréférencement** : c'est la possibilité d'accéder à l'entité (objet) référencé par un pointeur.

```

int i = 0;
int *pointeurSurEntier;
pointeurSurEntier = &i; // le pointeur contient l'adresse de i : acces indirect a l'entier
*pointeurSurEntier = *pointeurSurEntier + 1; // déréférencement du ponteur

```

En C++, il est possible de définir, en plus des pointeurs, des références.

Une référence est un synonyme d'un identificateur. Elle permet de manipuler une variable sous un autre nom que celui avec lequel elle a été déclarée.

```

int i;
int &ref = i;
i = ref + i; // double la valeur de i (et de ref)

```

Le passage de paramètres

L'en-tête de la définition d'une fonction contient en particulier une liste de paramètres de cette fonction : on parle de **paramètres formels**.

Lors de l'appel (ou invocation) d'une fonction, cet appel est réalisé en spécifiant des **paramètres d'appel** (on dit aussi **paramètres actuels** ou **paramètres effectifs**).

Pour qu'un appel soit correct, il faut tout d'abord que le nombre des paramètres effectifs soit égal à celui des paramètres formels (sauf si le langage autorise des paramètres optionnels). Par ailleurs, chaque paramètre effectif doit être d'un type compatible avec celui du paramètre formel correspondant (celui en même position dans la définition de la fonction).

Le compilateur Java impose le respect total de ces règles.

```
--> cat Param1.java
class Param1 {
    static float puis(float x, int n) {
        float y = 1;
        for(int i = 0; i < n; i++) y = y * x;
        return y; }
    public static void main(String[] args) {
        System.out.println(puis(9.81, 3)); }
}
--> javac Param1.java
Param1.java:8: puis(float,int)
      in Param1 cannot be applied to (double,int)
        System.out.println(puis(9.81, 3));
                          ^
1 error
-->
```

Le mode de passage des paramètres définit la manière dont un paramètre effectif est substitué lors d'un appel de fonction au paramètre formel correspondant.

Il y a deux modes principaux de passage (ou transmission) de paramètres (nous nous limitons volontairement à ces deux-là qui peuvent être illustrés au travers d'exemples Java, C ou C++).

- **le passage par valeur** : avec ce mode de passage de paramètre, le paramètre formel est vu comme une variable locale de la fonction (c'est-à-dire créée à l'entrée dans la fonction et supprimée à la sortie de la fonction et donc uniquement visible et accessible durant l'exécution du corps de la fonction) dont la valeur initiale est celle du paramètre effectif correspondant. Le paramètre effectif ne sert donc, au travers de sa valeur, qu'à permettre d'initialiser le paramètre correspondant de la fonction à sa création. Il n'y a aucun lien, en ce qui concerne les adresses, entre le paramètre effectif et le paramètre formel lors de l'exécution de la fonction. Un changement de valeur du paramètre formel sera perdu au retour de la fonction : le paramètre effectif correspondant ne sera pas modifié.

C'est ce mode de transmission des paramètres qui est utilisé en Java et en C. Cependant, compte tenu de la nature des variables en Java (dont les valeurs sont soit des valeurs d'un type primitif, soit des références sur un type non primitif) ou de la possibilité de définir des pointeurs en C, nous verrons dans le prochain cours, qu'il est possible, dans une fonction, de modifier un objet dont une référence (ou un pointeur) est transmise en paramètre ;

- **le passage par référence** (ou par adresse) : avec ce mode de transmission de paramètres, le paramètre effectif est une référence dont le paramètre formel devient un synonyme lors de l'appel. Le paramètre formel et le paramètre effectif sont alors deux références synonymes et désignent donc un même espace en mémoire. Toute modification du paramètre formel dans la fonction modifie donc son synonyme (le paramètre effectif).

En C++, le mode de passage par valeur est le mode par défaut. Cependant, il est possible, lors de l'écriture d'une fonction, de choisir individuellement pour chacun des paramètres s'il sera transmis par valeur ou par référence. Un tel choix est aussi possible par exemple en Pascal.

Dans les langages PL1 et Fortran, c'est le passage par adresse qui est utilisé.