

Programmation 1 bis

Références, listes chaînées

Gilles Trombettoni

IUT de l'Université de Montpellier, département informatique

Novembre-décembre 2017

Plan

- 1 Références
- 2 Passage de paramètres
- 3 Tableaux
- 4 Listes chaînées

Définition

- En programmation, une **référence** est une valeur qui permet l'accès en lecture et en écriture à une donnée, située généralement en mémoire principale.
- Une référence n'est pas la donnée elle-même mais seulement une information permettant de la localiser.
- La plupart des langages de programmation permettent l'utilisation de références, que ce soit de façon explicite (langage C) ou implicite (langage Java).

Deux grandes catégories de références

- Le type de référence le plus simple, mais aussi le plus dangereux, est le **pointeur**.
Il s'agit simplement d'une **adresse mémoire**.
- Les références sont souvent vues comme de simples **noms** (alias) identifiant une donnée, comme les objets et les tableaux en Java.

Dangers liés aux références

Plusieurs noms peuvent référencer la même donnée.
Conséquences :

Libération de la donnée/objet sans invalider ses références

⇒ **erreur de segmentation** lors d'une future utilisation

Suppression des références d'un objet qui devient inaccessible

⇒ **fuite de mémoire**

Dangers seulement au cas où le langage n'offre pas de mécanisme de ramasse-miettes (*garbage collector*). En C/C++ par exemple.

A retenir

- `int[] tab;` Déclaration/création d'une **référence** sur un tableau.
- `tab = new int[10];`
Création d'une zone mémoire référencée par `tab`.
- `Fraction frac;` Déclaration/création d'une **référence** sur un objet.
- `frac = new Fraction(-7,11);`
Création d'une zone mémoire référencée par `frac`.

```
public class TabToString {  
    public static void main(String args[]) {  
        int[] tab = new int [2];  
        tab[0]=-3; tab[1]=10;  
        System.out.println(tab.toString()); // affiche tab  
    }  
}  
  
> java TabToString  
[I@3487a5cc
```

Plan

- 1 Références
- 2 Passage de paramètres
- 3 Tableaux
- 4 Listes chaînées

Echange d'information avec les fonctions/méthodes

En Java, le passage de paramètres et le retour d'une méthode s'effectuent par **valeur**, par **copie**. Conséquences :

- Les paramètres de **type primitif** (`int`, `double`, `char`, etc) ne sont pas modifiés lors de l'appel.
- Les paramètres **objet ou tableau** peuvent être modifiés lors de l'appel.

En effet, le nom d'un objet ou d'un tableau correspond à une **référence** sur une donnée en mémoire, et non pas à la donnée elle-même : lors du passage de paramètre, on copie la référence.

Par conséquent, la donnée elle-même (non copiée) peut être modifiée à l'intérieur de la fonction.

Echange d'information avec les méthodes

En résumé, tout se passe comme si on avait :

- transmission par donnée pour les types primitifs ;
- transmission par donnée/résultat pour les objets et les tableaux.

Echange d'information avec les méthodes : primitifs

```
public class PrimitifArgument {  
    public static void main (String args[]) {  
        int n = 10;  
        System.out.println("Avant appel : " + n);  
        doubleValeur(n);  
        System.out.println("Après appel : " + n);  
    }  
    public static void doubleValeur (int a) {  
        System.out.println("Debut procedure : " + a);  
        a = 2*a;  
        System.out.println("Fin procedure : " + a);  
    }  
}
```

```
> java PrimitifArgument  
Avant appel      : 10  
Debut procedure  : 10  
Fin procedure    : 20  
Après appel      : 10
```

Echange d'information avec les méthodes : objets

```
public class ObjetArgument {  
    public static void main (String args[]) {  
        Fraction f = new Fraction(-2, 3);  
        System.out.print("Fraction avant : " + f);  
        modifieFraction(f, 5, 8);  
        System.out.print("Fraction apres : " + f);  
    }  
    public static void modifieFraction (Fraction frac,  
                                         int num, int denom) {  
        frac.setFraction(num, denom);  
    }  
}
```

```
> java ObjetArgument  
Fraction avant : -2/3  
Fraction apres : 5/8
```

Echange d'information avec les méthodes : tableaux

```
public class TableauArgument {  
  
    public static void main (String args[]) {  
        double T[] = {3.5, -2.0, 8.6, 3.14};  
        System.out.print("Tableau T avant : ");  
        Utile.affiche(T); // affichage du tableau  
        Utile.raz(T);      // remise à zéro du tableau  
        System.out.print("Tableau T apres : ");  
        Utile.affiche(T);  
    }  
}
```

```
> java TableauArgument  
Tableau T avant : 3.5 -2.0  8.6  3.14  
Tableau T apres : 0.0  0.0  0.0  0.0
```

Echange d'information avec les méthodes : tableaux

```
public class Utile {  
  
    static void raz (double tab[]) {  
        for (int i = 0; i < tab.length ; i++) {  
            tab[i] = 0.0;  
        }  
    }  
  
    static void affiche (double tab[]) {  
        for (int i = 0; i < tab.length ; i++) {  
            System.out.print(tab[i] + " ");  
        }  
    }  
}
```

Plan

- 1 Références
- 2 Passage de paramètres
- 3 Tableaux**
- 4 Listes chaînées

Rappel sur les tableaux

Structure de données composite dont tous les éléments sont contigus en mémoire et sont du même type.

<code>double[] tab1, tab2;</code>	tab1 et tab2 contiennent la référence d'un tableau de <code>double</code>
<code>double tab1[], x, tab2[];</code>	idem
<code>tab1 = new double[3];</code>	allocation de 3 elts (0.0) en mémoire
<code>int tab3[4];</code>	erreur de compilation
<code>double[] tab1 = {3.5, x+y};</code>	initialisation avec 2 éléments
<code>tab1[0] = 6.7;</code>	affectation (accès en écriture)
<code>...print(tab1[0]+tab1[1]);</code>	accès en lecture

- Les tableaux sont créés à l'exécution (par le programme `java`).
- Ils ne peuvent pas changer de taille une fois créés.
- La même référence (`tab`) peut changer de tableau.

```
System.out.println("Taille de tableau souhaitée ?");  
Scanner sc = new Scanner(System.in); int taille=sc.nextInt();  
double tab[] = new double[taille];
```

Tableaux à plusieurs indices

- Les tableaux multi-dimensionnels sont des tableaux de tableaux.
- On peut donc déclarer : `int t[][]={{1,2,3},{100,200}}`.
Question 1 : Dessiner en mémoire ce tableau.

- Question 2 : Qu'affiche les lignes suivantes ?

```
int t2[][] = t; t2[1][1]=-10;  
System.out.println(t[1][1]);
```

- Question 3 : Dessiner en mémoire le tableau t :

```
int t[][]; t = new int[2][];  
int[] t1 = new int[3]; int[] t2 = new int[2];  
t[0] = t1; t[1] = t2;
```

Copie superficielle versus copie profonde

```
Fraction[] t1 = {new Fraction(2,3), new Fraction(4,5)};
```

```
Fraction[] t2 = t1; // copie très superficielle
```

```
Fraction[] t3 = new Fraction[2];  
for (int i = 0; i < 2; i++) { // copie superficielle  
    t3[i] = t1[i];  
}
```

```
Fraction[] t4 = new Fraction[2];  
for (int i = 0; i < 2; i++) { // copie profonde  
    // appel au constructeur par copie:  
    t4[i] = new Fraction(t1[i]);  
}
```


Plan

- 1 Références
- 2 Passage de paramètres
- 3 Tableaux
- 4 Listes chaînées

Listes chaînées : définition

Une **liste chaînée** est une suite de couples formés d'un élément et d'une référence vers l'élément suivant (jeu de piste).



Avantages et inconvénients par rapport à un tableau

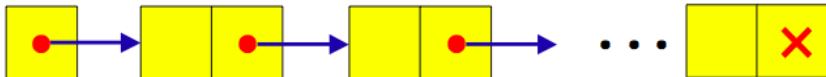
- Pour : redimensionnement de la taille d'une liste (ajout d'éléments)
- Pour : ajout et retrait d'un élément (référencé) en temps constant
- Contre : accès en temps linéaire au dernier élément de la liste
- (Contre : plus de place en mémoire pour le même nombre d'éléments)

Listes chaînées : déclaration

```
public class Maillon {  
  
    private int valeur;  
    private Maillon suivant;  
  
    public Maillon () {this.suivant = null;} // Constr. vide  
    public Maillon (int n) { // Constructeur avec une valeur  
        this.valeur = n; this.suivant = null;  
    }  
  
    /* Accesseurs */  
    public int getVal() { return this.valeur; }  
    public void setVal(int v) { this.valeur = v; }  
    public Maillon getSuiv () { return this.suivant; }  
    public void setSuiv (Maillon m) {  
        this.suivant = m;  
    }  
}
```

Listes chaînées : déclaration

```
public class Liste {  
  
    private Maillon tete; // seul attribut !  
  
    /** Constructeur d'une liste vide */  
    public Liste () { this.tete = null; }  
  
    /** Constructeur d'une liste a un seul element */  
    public Liste (int n) {  
        this.tete = new Maillon(n);  
    }  
    /** "Accesseur" en lecture */  
    public int valTete () { return this.tete.getVal(); }  
}
```



Opérations de base

```
public class Liste {  
    ...  
    public boolean estVide() {  
        return (this.tete == null);  
    }  
  
    public void ajoutTete (int n) {  
        Maillon m = new Maillon(n);  
        m.setSuiv(this.tete); // m.suivant = this.tete;  
        this.tete = m;  
    }  
    ...  
}
```



Opérations de base

...

```
public boolean contient (int n) {
    Maillon courant = this.tete;
    while (courant != null && courant.getVal() != n) {
        System.out.println("hello");
        courant = courant.getSuiv();
    }
    return courant != null;
}
...
```

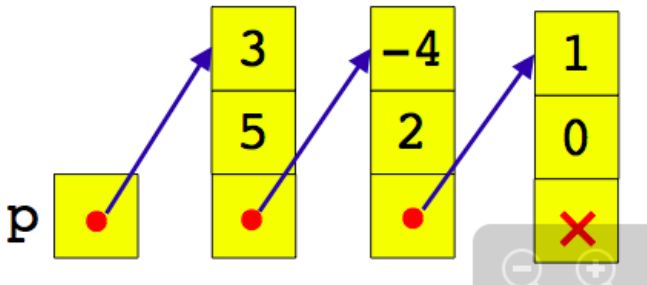


Exemple

```
public class MainListe {  
  
    public static void main (String args[]) {  
        Liste L = new Liste(3);  
        L.ajoutTete(2);  
        L.ajoutTete(5);  
        System.out.println("Tete de L = "  
                             + L.valTete());  
        System.out.println("2 appartient a L ? "  
                             + L.contient(2));  
    }  
}  
  
> java MainListe  
Tete de L = 5  
hello  
2 appartient a L ? true
```

Polynômes représentés par des listes

Exemple du polynôme $3x^5 - 4x^2 + 1$ (dessins de Jean-Eric Pin)



Exemple de représentation par liste

```
public class Poly {
    private Monome tete; // attribut
    public Poly (int coef, int degre) {
        this.tete = new Monome(unCoef,unDegre);
    }
    public void ajoutMonomeTete (int unCoef, int unDegre) {
        Monome newMonome = new Monome(unCoef, unDegre);
        newMonome.setSuiv(this.tete); // newMonome.suiv = this.tete
        this.tete = newMonome;
    }
}

public class Monome {
    private int coef, degre;
    private Monome suivant;

    public Monome (int unCoef, int unDegre){
        this.coef = unCoef; this.degre = unDegre; this.suivant = null;
    }
    public void setSuiv (Monome m) { this.suivant = m; }
    public Monome getSuiv () { return this.suivant; }
    ...
}
```

Affichage d'un polynôme

```
public String toString() { // Dans classe Monome
    if (this.degre > 1)
        return coef + "x^" + degre;
    else if (this.degre == 1)
        return coef + "x";
    else // degre == 0
        return String.valueOf(this.coef);
}

public String toString() { // Dans classe Poly
    String s = ""; Monome monome = this.tete;
    while (monome != null) {
        String strMono = monome.toString();
        if (monome.getSuiv() != null) strMono += " + ";
        s += strMono;
        monome = monome.getSuiv();
    }
    return s;
}
```

Création et affichage d'un polynôme

```
public class MainPol {  
  
    public static void main(String[] args) {  
  
        Poly p = new Poly(-2, 0);  
        p.ajoutMonomeTete(5, 1);  
        p.ajoutMonomeTete(4, 3);  
        p.ajoutMonomeTete(-6, 108);  
  
        System.out.println(p);  
    }  
}
```

affiche: $-6x^{108} + 4x^3 + 5x + -2$