Introduction à la Programmation : Fonctions/méthodes «auxiliaires» (réutilisabilité et modularisation)

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs du cours d'aujourd'hui

- Introduire la notion de **réutilisabilité**
- Présenter les **méthodes auxiliaires** en Java
  - Définition de méthodes auxiliaires
  - Appel d'une méthode et passage des arguments
  - Surcharge de méthodes

Support MOOC

Vidéo et Quiz de :

https://www.coursera.org/learn/initiation-programmation-java/home/week/6

- Semaine 6
- Notes de cours et BOOC, semaine 6

Pendant l'heure de cours

- Petit rappel des points importants
- Fiches résumé : fonctions
- Approfondissements :
  - Etude de cas 1 : implications du passage par valeur systématique (à découvrir en cours)
  - Ellipses et arguments optionnels(41)
  - Etude de cas 2 (calcul de l'IMC revisité, à découvrir en cours)

La notion de réutilisabilité

Jusqu'à présent : **programme = une classe et une méthode**

```
class Scores {
   public static void main(String[] args){
      Scanner keyb = new Scanner(System.in);
      System.out.print("Donnez le nombre de joueurs:");
      int n = keyb.nextInt();
      int moyenne = 0;
      int[] scores = new int[n];
     // calcul de la moyenne
      for (int i = 0; i < n; i++) {
         System.out.println("Score joueur " + i + " :");
         scores[i] = keyb.nextInt();
         moyenne = moyenne + scores[i]; }
      if (n != 0)
         moyenne = moyenne / n;
     // Affichage des scores
      System.out.println(" Score " + " Ecart Moyenne");
      for (int i = 0; i < n; i++) {
         System.out.println(scores[i] + " "
                        + (scores[i] - moyenne));
      }
   }
}
```

La notion de réutilisabilité (2)

Supposons maintenant que l'on ait à réaliser les mêmes traitements pour **deux jeux différents** (avec des ensembles de joueurs différents)

```
int moyenneJeu1 = 0;
// n1 : nombre de joueurs du premier jeu
int[] scoresJeu1 = new int[n1];
int moyenneJeu2 = 0;
// n2 : nombre de joueurs du second jeu
int[] scoresJeu2 = new int[n2];

// Cumul des scores du premier jeu
for (int i = 0; i < n1; i++) {
   scoresJeu1[i] = keyb.nextInt();
   moyenneJeu1 += scoresJeu1[i]; }
// moyenne du premier jeu
if (n1 != 0)
   moyenneJeu1 /= n1;

// Cumul des scores du second jeu
for (int i = 0; i < n2; i++) {
   moyenneJeu2[i] = keyb.nextInt();
   moyenneJeu2 += scoresJeu2[i]; }
// moyenne du second jeu
if (n2 != 0)
   moyenneJeu2 /= n2;
//...
```

Des **portions identiques de code** devront être **dupliquées** à plusieurs endroits !

La notion de réutilisabilité (3)

Problème avec le code précédent :

- Pas de **partage** des parties importantes ou utilisées plusieurs fois

Par exemple, la tâche `P` de cumul des scores :

```
for (int i = 0; i < n; i++) {
    scores[i] = keyb.nextInt();
    moyenne += scores[i];
}
```

doit être exécutée **plusieurs fois**

Solution possible : **recopier** `P` autant de fois que nécessaire aux endroits appropriés

... c'est évidemment une **très mauvaise solution** !

Conseil : Ne **jamais** faire de copier/coller en programmant.

La notion de réutilisabilité (4)

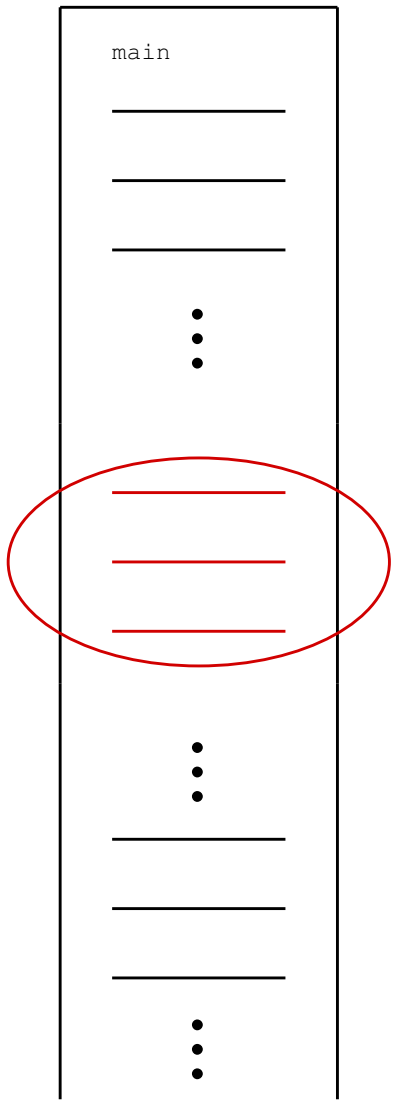
Pourquoi ne jamais dupliquer du code (copier/coller) :

- cela rend la **mise à jour** de ce programme plus **difficile** : reporter chaque modification de `P` dans chacune des copies de `P`
- cela **réduit** fortement **la compréhension** du programme résultant
- cela augmente inutilement la taille du programme

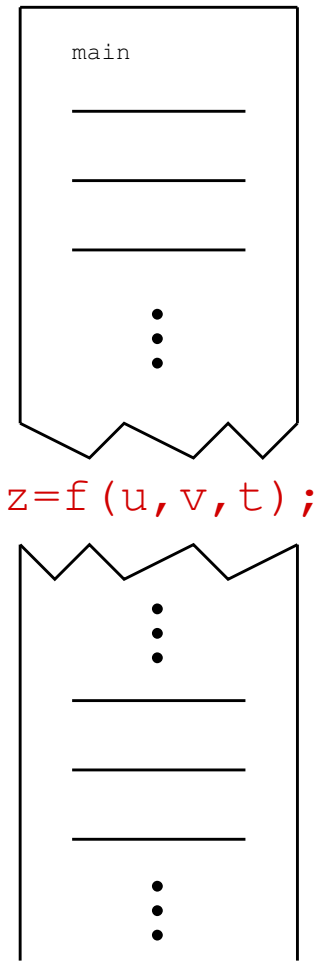
Comment faire alors pour répéter les mêmes traitements sans dupliquer le code ?

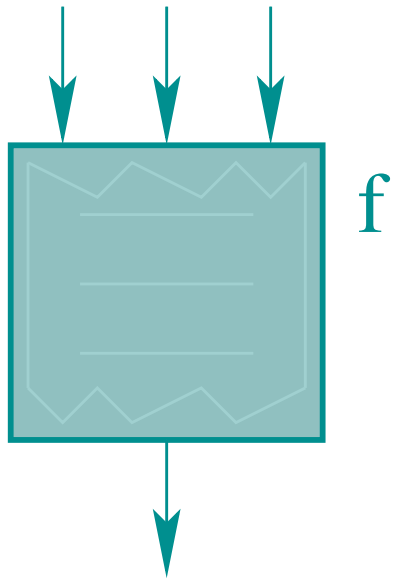
☞ Il faut modulariser les programmes en utilisant des **méthodes auxiliaires**

Notion de réutilisabilité (5)



Notion de réutilisabilité (5)





Méthodes

Méthode = portion de programme réutilisable

Plus précisément, une méthode est un construction logicielle caractérisée par :

un nom référence à l'objet "*méthode*" lui-même, indiquée lors de sa création

des arguments (les "*entrées*") ensemble de références à des objets définis à l'extérieur de la méthode dont les valeurs sont potentiellement utilisées dans le corps de la méthode

un corps le code à réutiliser qui a justifié la création de la méthode

des variables internes (spécifiques au corps de la méthode) : notion de **portée locale**.

une valeur de retour (la "*sortie*", pas obligatoire) indiquée, le cas échéant par la commande `return`

Exemples

Exemple 1 : La méthode `main`

```
public static void main (String[] args) {
    //instructions
    // ....
}
```

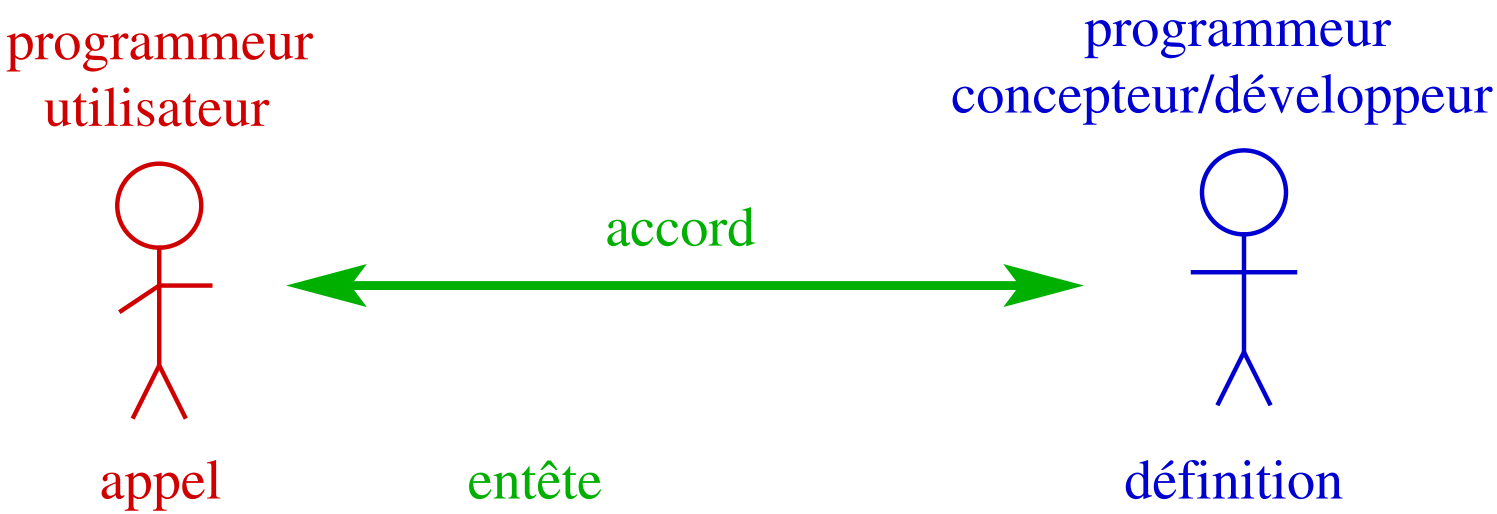
Exemple 2 : Nos méthodes d'Entrée-Sortie

```
System.out.println("Bonjour");
Scanner keyb = new Scanner(System.in);
int a = keyb.nextInt();
```

- Une instruction est souvent un appel à une autre méthode
- Le code de la méthode `keyb.nextInt` n'est pas présent dans la méthode `main`
- Il est défini ailleurs afin de :
  - raccourcir la méthode appelante (`main`)
  - être utilisable depuis plusieurs endroits

Les « 3 facettes » d'une méthode

- Résumé / Contrat (« entête »)
- Création / construction (« définition »)
- Utilisation (« appel »)



Méthodes auxiliaires

Petit résumé sur les méthodes auxiliaires :

- Elles permettent de regrouper des traitements partagés ou répétés (réutilisabilité).
- Elles peuvent être utilisées par d'autres méthodes.
- Leur déclaration est précédée du mot clé `static` (pour le moment). Les méthodes auxquelles on associe le mot clé `static` sont aussi appelées **méthodes de classes** (expliqué dans un prochain cours. . .).

Comment modulariser ?

Afin de modulariser un programme :

1. Localiser une suite d'instructions qui représente une tâche délimitée
2. Définir une méthode à part :
   - En-tête qui identifie la méthode
   - Instructions (à transférer depuis le programme à modulariser)
3. Appeler la méthode depuis l'endroit où se trouvaient les instructions transférées
4. Affectation de la valeur retournée
   (le cas échéant)

Exercice : Modularisons notre classe `Scores`

Plan de Modularisation

Commençons par identifier des tâches délimitées :

- Lecture des tableaux de notes
- Calcul des moyennes
- Affichage des tableaux de notes

Exemple : Méthode `main` revisitée :

```
public static void main (String[] args) {
    int n1 = nbJoueurs();
    int n2 = nbJoueurs();
    // remplissage des tableaux de scores
    int[] scoresJeu1 = lireScores(n1);
    int[] scoresJeu2 = lireScores(n2);
    // calcul des moyennes
    int moyenneJeu1=calculerMoyenne(scoresJeu1);
    int moyenneJeu2=calculerMoyenne(scoresJeu2);
    // affichages
    afficherScores(scoresJeu1, moyenneJeu1);
    afficherScores(scoresJeu2, moyenneJeu2);
}
```

On duplique toujours un peu, mais c'est beaucoup plus concis et élégant !

☞ Exercice : Faites mieux avec un tableau de jeux

Déclaration des méthodes auxiliaires

Syntaxe générale :

```
static type nom(type1 nom1, type2 nom2, ...) {
   ...      // instructions
}
```

- **`static`**
  - Mot-clé obligatoire (pour l'instant)
  - Explication simplifiée : méthode auxiliaire de la méthode `main`
- **`type`** est le type de la valeur retournée
  - `double`, `int`, `boolean`, `void`, ...
  - `void` signifie que la méthode ne retourne pas de valeur
- **`nom`** de la méthode
  - Identificateur
  - Commence par une minuscule : `lireScores`

Déclaration des méthodes auxiliaires (2)

Syntaxe générale :

```
static type nom(type1 nom1, type2 nom2, ...) {
  ...       // instructions
}
```

- `(type1 nom1, type2 nom2, ...)`
  - Liste des arguments (paramètres formels)
  - Indique le type et le nom de chaque argument
  - **Argument** = variable utilisable à l'intérieur de la méthode auxiliaire et destinée à contenir une valeur fournie par la méthode appelante.
- `instructions` : les instructions à exécuter

Où placer la déclaration d'une méthode ?

☞ N'importe où dans la classe, en dehors des autres méthodes

Exemple : `lireScores`

Codons la méthode auxiliaire qui lit et retourne un tableau de scores

- Argument : la taille du tableau.
- Valeur retournée : un tableau de **`int`** (lus du clavier)

Déclaration :

```
static int[] lireScores(int n) {
    Scanner keyb = new Scanner(System.in);
    int[] scores = new int[n];
    for (int i = 0; i < n; i++) {
        System.out.println("Score[" + i + "]: ");
        scores[i] = keyb.nextInt();
    }
    return scores;
}
```

Note : l'instruction `return` **retourne la valeur** résultant de l'appel de la méthode.

Exemple : `lireScores` (2)

A quel endroit peut-on invoquer une méthode auxiliaire comme `lireScores` ?

- A n'importe quel endroit où l'on a besoin d'une valeur dont le type correspond à celui de la valeur retournée par la méthode.

Exemples :

- dans une affectation :

```
int[] mesScores = lireScores(n);
```

```
double[] series = lireNotes(n);
```

- comme paramètre d'appel d'une autre méthode :

```
afficherScores(lireScores(n1), moyenneJeu1);
```

Méthodes sans arguments

Il est bien sûr possible de définir des méthodes **sans arguments**.
Il suffit, dans la définition, d'utiliser une liste d'arguments vide `()`.
Exemple :

```
class Exemple {
   public static void main(String[] args) {
     int n1 = nbJoueurs();
     int n2 = nbJoueurs();
     int [] scoresJeu1 = lireScores(n1);
     int [] scoresjeu2 = lireScores(n2);
//...
   }
   //  Declaration de le methode nbJoueurs
   static int nbJoueurs() {
     System.out.println("Donnez le nombre de joueurs :");
     Scanner keyb = new Scanner(System.in);
     int i = keyb.nextInt();
     return i;
   }
}
```


J. Sam

Méthodes sans valeur de retour

Il est aussi possible de définir des méthodes **sans valeur de retour**, c'est-à-dire des méthodes **qui ne renvoient rien**.

Il suffit, dans la définition, d'utiliser le type prédéfini `void` comme type de retour.

Exemple :

```
static void afficherScores(double[]scores,double moyenne){
   System.out.println("Score " + " Ecart a la Moyenne");
   for (int i = 0; (i < scores.length); i++) {
      System.out.println(scores[i] + " "
                         + (scores[i] - moyenne));
   }
   // Pas de return !
}
```

Remarques sur l'instruction `return`

1. Le type de la valeur retournée doit correspondre au type dans l'en-tête :

```
static double bidon() {
    boolean b = true;
    return b; // Erreur
}
```

2. `return` doit être la toute dernière instruction exécutée :

```
static double lire() {
    System.out.print("Entrez un nombre: ");
    Scanner keyb = new Scanner(System.in);
    double n = keyb.nextDouble();
    return n;
    System.out.println(); // Erreur
}
```


J. Sam

Remarques sur l'instruction return (2)

3. Le compilateur doit être sûr de toujours pouvoir exécuter un `return` :

```
static double lire() {
    System.out.print("Entrez un nombre: ");
    Scanner keyb = new Scanner(System.in);
    double n = keyb.nextDouble();
    if (n > 0){
        return n; // Erreur
    }
}
```

Méthodes et portée

1. *Les arguments d'une méthode* constituent des *variables locales* au corps de la méthode ;

2. Les variables déclarées dans le corps d'une méthode sont également *locales* à ce corps.

Plan

- Notion de réutilisabilité
- Définition de méthodes auxiliaires
- Appel d'une méthode et passage des arguments
- Surcharge de méthodes

Évaluation d'un appel de méthode

Pour une méthode définie par :

```
static int m(type1 x1, type2 x2,..., typeN xN)
                        { ... }
```

l'**évaluation** de l'appel

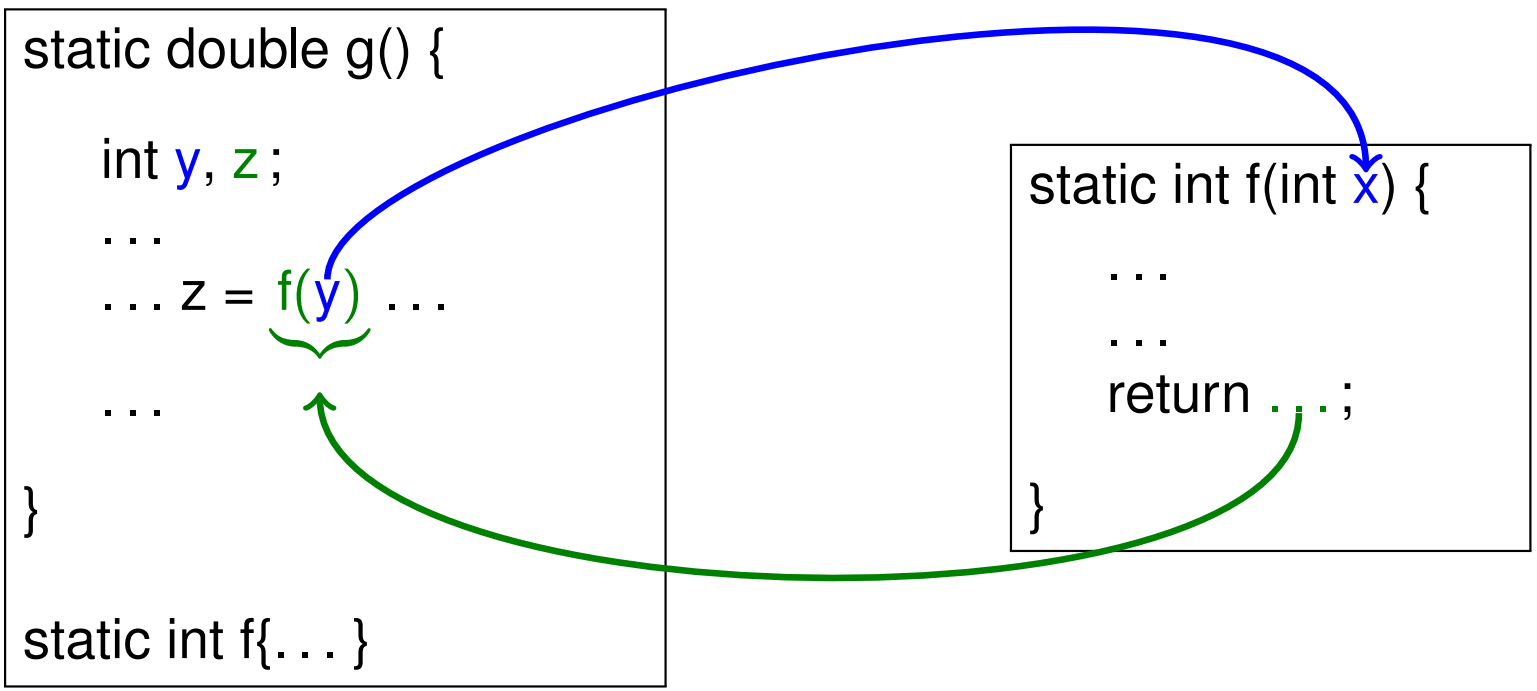
```
m(param1, param2,..., paramN)
```

s'effectue de la façon suivante :

1. les paramètres effectifs `param1`, `param2`, ..., `paramN` sont évalués et les valeurs produites sont **affectées** aux arguments `x1`, `x2`, ..., `xN` de la méthode `m`
   Concrètement, cette 1ère étape revient à faire : `x1 = param1`, `x2 = param2`, ..., `xN = paramN`
2. le programme correspondant au corps de la méthode `m` est exécuté
3. l'expression après la commande `return` est évaluée et retournée comme résultat de de l'appel.

Évaluation d'un appel de méthode (2)

L'évaluation de l'appel de méthode peut être schématisé de la façon suivante :



Le passage des arguments

Soit la situation suivante (pseudo-code) :

```
void methode(Type v){
  // traitement modifiant v
}

// ailleurs, dans le programme principal,
// par exemple:
Type v1 = .. ; // initialisation de v1
methode(v1);
// v1 EST-ELLE MODIFIEE ICI OU NON???
```

En programmation de façon générale, on dira que :

- L'argument `v` est **passé par valeur** si `methode` ne peut pas modifier `v1` : `v` est une **copie locale** de `v1`.
- L'argument `v` est **passé par référence** si `methode` peut modifier `v1`

En Java, il n'existe que le passage par valeur : `v` est **toujours une copie** de `v1`.

☞ La réponse à la question dans le code est donc NON ! !

Le passage des arguments (2)

MAIS.. puisque Java ne manipule pas les types élémentaires comme les types évolués, on peut se poser la question sous un autre angle :

```
//...
void methode(Type v){ // Type :type EVOLU'E
// traitement modifiant l'objet referenc'e par v
// traitement modifiant v lui meme
}
// ailleurs:
Type v1 = ..; // initialisation de v1
methode(v1);
//1. v1 est-elle modifi'ee ici?
//2. l'objet referenc'e par v1 est-il modifi'e
//   ici?
```

- On a toujours un passage par valeur car la référence `v` est une copie de `v1`.
- Cependant, `Type` étant évolué, l'argument qui est donné à `methode` lors de l'appel `methode(v1)` est une copie de la référence à `v1` (son adresse) : l'objet pointé par `v` est le même que l'objet pointé par `v1`. **Toute modification faite sur l'objet référencé via `v` est donc visible via `v1` !**

Le passage des arguments (2)

MAIS.. puisque Java ne manipule pas les types élémentaires comme les types évolués, on peut se poser la question sous un autre angle :

```
//...
void methode(Type v){ // Type :type EVOLU'E
// traitement modifiant l'objet referenc'e par v
// traitement modifiant v lui meme
}
// ailleurs:
Type v1 = ..; // initialisation de v1
methode(v1);
//1. v1 est-elle modifi'ee ici?
//2. l'objet referenc'e par v1 est-il modifi'e
//   ici?
```

☞ La réponse à la question 2 est donc OUI (et reste non pour la question 1)

Type de base : exemple typique du passage par valeur

Exercice : Qu'affiche le code suivant ?

```
public static void main(String[] args) {
    int val = 1;
    m(val);    // TYPE DE BASE
               // PASSAGE PAR VALEUR
    System.out.println(" val= " + val);
}

static void m(int x) {
    x = x + 1;
    System.out.print("x= " + x);
}
```

Réponse : `x=2 val=1`

Les modifications effectuées à l'intérieur de la méthode `m()` ne se répercutent pas sur la variable `val` associée à l'argument `x` passé par valeur.

Type évolué : modification de la référence

Exercice : Qu'affiche le code suivant ?

```
public static void main(String[] args) {
    int[] tab = {1};
    m(tab); // tab (reference)
            // PASSAGE PAR VALEUR AUSSI
    System.out.println(" tab[0]= " + tab[0]);
}
static void m(int[] x) {
    int[] t = {100};
    x = t; //Modification de la reference
           //(on met une autre adresse dans x)
    System.out.print("x[0]= " + x[0]);
}
```

Réponse :

```
x[0]= 100 tab[0]= 1
```

les modifications faites dans la méthode
sur la référence elle-même ne sont pas visibles à l'extérieur de la méthode !

Il y aura un exemple avec `String` pendant le TP !

Type évolué : modification de l'objet référencé

Exercice : Qu'affiche le code suivant ?

```
public static void main(String[] args) {
    int[] tab = {1};
    m(tab);
    System.out.println(" tab[0]= " + tab[0]);
}
static void m(int[] x) {
    x[0] = 100; // modification de l'objet
                // referenc'e par x
    System.out.print("x[0]= " + x[0]);
}
```

Réponse : `x[0]= 100 tab[0]= 100`

les modifications faites dans la méthode sur l'objet référencé restent visibles à l'extérieur de la méthode ! (on a copié dans `x` la référence `tab` : `x` et `tab` pointent sur le même tableau.)

Le passage des arguments

Type évolué : autre exemple

Modifions la méthode :

```
static double[] lireScores(int n) {
    int[] scores = new int[n];
    for (int i = 0; i < n; i++) {
        System.out.println("Score[" + i + "]: ");
        scores[i] = keyb.nextInt();
    }
    return scores;
}
```

de sorte à ce qu'elle prenne le tableau `scores` en argument.

```
static void lireScores(int[] scores) {
    // scores est une REFERENCE
    for (int i = 0; i < scores.length; i++) {
        System.out.println("Score[" + i + "]: ");
        scores[i] = keyb.nextInt();
    }
}
```

Soit `mesScores` un tableau déclaré dans la méthode appelante.

`lireScores(mesScores)` **modifiera directement** ce tableau en mémoire !

Le passage des arguments : résumé

Type de base

La variable locale associée à l'argument correspond à une **copie** de l'argument (*i.e.*, un objet distinct mais de même valeur littérale).

- ☞ Les modifications effectuées à l'intérieur de la méthode **ne sont pas répercutées à l'extérieur** de la méthode

Le passage des arguments : résumé

Type évolué

La variable locale associée à l'argument correspond aussi à une copie de la **référence** sur l'objet qui lui est associé lors de l'appel.

- ☞ Les modifications effectuées sur la référence à l'intérieur de la méthode **ne sont pas répercutées à l'extérieur** de la méthode
- ☞ Les modifications effectuées sur l'objet référencé à l'intérieur de la méthode **sont répercutées à l'extérieur** de la méthode

Portée et appel de méthodes : Exercice

Dans le programme suivant :

```
public static void main(String args[]) {
    int i = 2
        m(i);
}

static void m(int a) {
    int i = 10;
    System.out.println(a * i);
}
```

À quel objet fait référence `i` dans l'instruction `System.out.println(a * i)` ?

Plan

- Notion de réutilisabilité
- Définition de méthodes auxiliaires
- Appel d'une méthode et passage des arguments
- Surcharge de méthodes

La surcharge de méthodes

En Java, les types des arguments font partie intégrante de la définition d'une méthode

signature d'une méthode = son nom + ses arguments et leurs types

Surcharge des méthodes :

- plusieurs méthodes peuvent avoir le même nom
- ... à condition qu'elles n'aient pas les mêmes listes d'arguments : nombre ou types d'arguments différents.

Le mécanisme de **surcharge des méthodes**, est très utile pour écrire des méthodes "*sensibles*" au type de leurs arguments, c'est-à-dire des méthodes correspondants à des traitements de même nature mais s'appliquant à des entités de types différents.

La surcharge de méthodes (2)

Exemple :

```
static void affiche(int x) {
    System.out.println("entier : " + x ) ;
}
static void affiche(double x) {
    System.out.println("reel : " + x ) ;
}
static void affiche(int x1, int x2) {
    System.out.println("paire : " + x1 + x2 ) ;
}
```

Les appels **`affiche(1)`**, **`affiche(1.0)`** et **`affiche(1,1)`** produiront alors les affichages différents suivants :

```
entier : 1
réel : 1.0
paire : 11
```

Ellipses

Il est possible de définir des méthodes avec un nombre variables d'arguments.
Syntaxe : `Type1 method(Type2... param)`

Les trois points (tournure elliptique) font partie de la syntaxe.

Ellipses : exemple

Exemple :

```
double moyenne(double... valeurs)
{
        if (valeurs.length == 0){
                return 0.0;
        }
        double somme = 0;
        for (double val : valeurs){
                somme += val;
        }
        return somme/valeurs.length;
}
```

Appels : `moyenne()`, `moyenne(6.0, 4.5, 2.5)`

Ellipses

Il ne peut y avoir qu'une ellipse par entête

D'autres paramètres sont possibles en plus mais l'ellipse doit être le dernier argument de l'entête

Les méthodes auxiliaires

Définition :

```
static type nom(type1 arg1, ..., typeN argN)
{
   //corps
   return value;
}
```

`type` est `void` si la méthode ne retourne aucune valeur.

Passage par valeur :
`type m(typedeBase arg);`
`m(x)` → `x` ne peut pas être modifié par `m`

Passage d'une **référence** (par valeur aussi) :
`type m(typeObjet arg);`
`m(x)` → la référence `x` ne peut pas être modifiée par `m`
`m(x)` → **l'objet référencé** par `x` peut être modifié par `m`

Surcharge (exemple) :

```
void affiche (int arg);
void affiche (double arg);
void affiche (int arg1, int arg2);
```

Pour résumer

Méthodologie pour construire une méthode

1. clairement identifier ce que **doit faire** la méthode
   ☞ ne pas se préoccuper ici du *comment*, mais bel et bien du **quoi** !
   (ce point n'est en fait que conceptuel, on n'écrit aucun code ici !)
2. que doit recevoir la méthode pour faire cela ?
   ☞ identifie les **arguments** de la méthode
3. Choisir des **noms pertinents** pour la méthode et ses paramètres
   Cela augmente la lisibilité de votre code (et donc facilite sa maintenance).
4. que doit « retourner » la méthode ☞ type de retour
   Se poser ici la question (pour une méthode nommée `f`) :
   est-ce que cela a un sens d'écrire :
   ```
   z = f(....);
   ```
   Si oui ☞ le type de `z` est le type de retour de `f`
   Si non ☞ le type de retour de `f` est `void`
5. (maintenant, et seulement maintenant) se préoccuper du *comment* :
   c'est-à-dire comment faire ce que doit faire la méthode ?
   c'est-à-dire écrire le corps de la méthode

Ce que j'ai appris aujourd'hui

- A développer des portions de code **réutilisables** : les méthodes auxiliaires
  - définition
  - arguments : le passage par **valeur** (et ses implications lorsque l'argument est une **référence**)

☞ je peux maintenant "*modulariser*" mes programmes, créer des composants logiciels **réutilisables**

Pour le prochain cours

Pas de nouvelles vidéos à regarder.

Le prochain cours :

- Présentation du mini-projet

J. Sam