Cours d'introduction à la programmation (en Java)
Fonctions/Méthodes

Jamila Sam
Jean-Cédric Chappelier
Vincent Lepetit

Faculté I&C

Notion de réutilisabilité

Pour l'instant : un programme est une séquence d'instructions

☞ mais sans **partage** des parties importantes ou utilisées plusieurs fois

Si une tâche, par exemple :

```
do {
    System.out.println("Entrez un nombre entre 1 et 100 : ");
    i = clavier.nextInt();
} while ((i < 1) or (i > 100));
```

doit être exécutée à *plusieurs* endroits dans un plus gros programme

☞ recopie ? **NON !**

Bonne pratique : Ne *jamais dupliquer* de code en programmant :
Jamais de « copier-coller » !

☞ Ce que vous voudriez recopier doit être mis dans une **fonction**

Notion de réutilisabilité (2)

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

Cela rend le programme

- inutilement long
- difficile à comprendre
- difficile à **maintenir** :
  reporter chaque modification dans *chacune* des copies

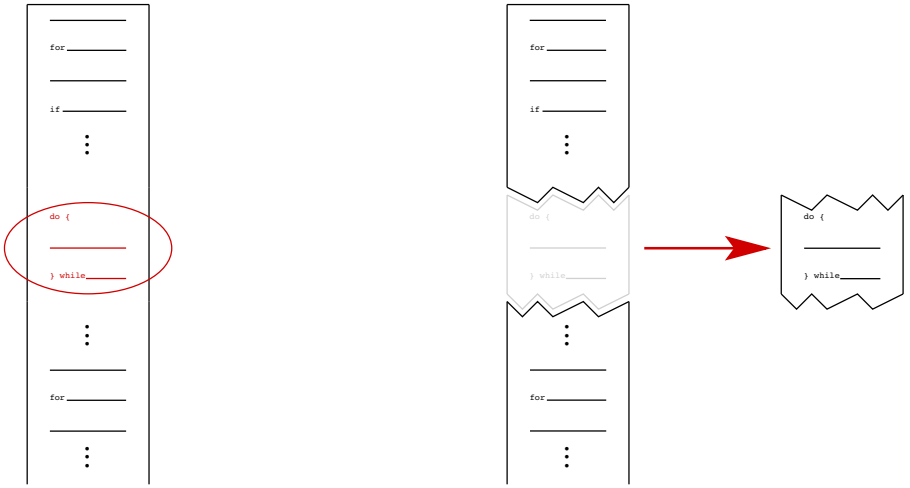
Tout bon langage de programmation fournit donc des moyens pour permettre la **réutilisation** de portions de programmes.

☞ les **fonctions**

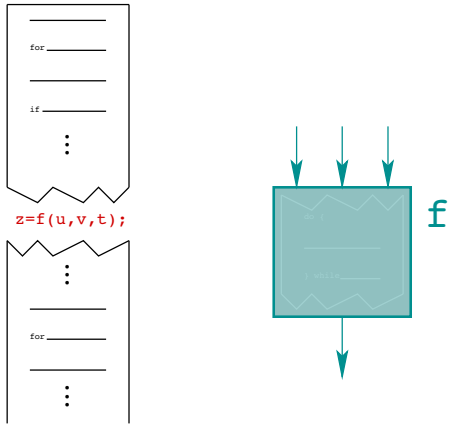
Exemple de fonction

```
int score (double points, double tempsJeu)
{
 int leScore = 0;
 if (tempsJeu != 0) {
  leScore = points / tempsJeu;
 }
 return leScore;
}
```

Notion de réutilisabilité : illustration



Notion de réutilisabilité : illustration



Fonction (en programmation)

fonction = portion de programme réutilisable ou importante en soi

Plus précisément, une fonction est un objet logiciel caractérisé par :

- un corps : la portion de programme à réutiliser ou mettre en évidence, qui a justifié la création de la fonction ;
- un nom : par lequel on désignera cette fonction ;
- des paramètres : (les « *entrées* », on les appelle aussi « *arguments* ») ensemble de variables extérieures à la fonction dont le corps dépend pour fonctionner ;
- un type et une valeur de retour : (la « *sortie* ») ce que la fonction renvoie au reste du programme

L'utilisation de la fonction dans une autre partie du programme se nomme un appel de la fonction.

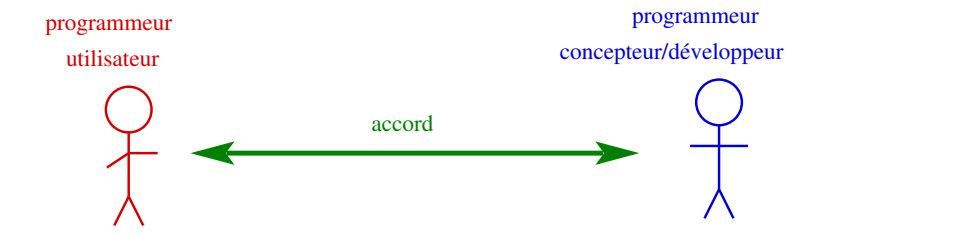
Terminologie

Dans les langages uniquement orienté-objet, comme c'est le cas de Java, le terme de « *méthode* » est généralement utilisé à la place de celui de « fonction ».

☞ C'est ce terme que nous utiliserons désormais

Les « 3 facettes » d'une méthode

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



appel

```
z = f(2*u, v+3);
```

entête

```
double f(double x,
         double y)
```

définition

```
double f(double a, double b)
{
  ...
}
```

Exemple complet

```
class Exemple
{
 private static Scanner clavier= new Scanner(System.in);
 public static void main(String[] args)
  {
   double note1 = 0.0;
   double note2 = 0.0;
   System.out.println("Entrez vos deux notes : ");
   note1 = clavier.nextDouble();
   note2 = clavier.nextDouble();
   System.out.println("Votre moyenne est : "
          + moyenne(note1, note2));
  }

  static double moyenne(double x, double y)
{
   return (x + y) / 2.0;
}
```

appel

entête
définition

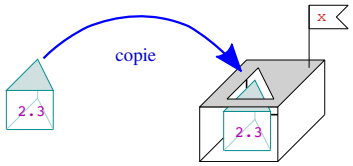
Évaluation d'un appel de méthode

```
double moyenne (double x, double y)
{
  return (x + y) / 2.0;
}
```

Que se passe-t-il lors de l'appel suivant :

```
z = moyenne( 1.5 + 0.8, 3.4 * 1.25 );
```

1. évaluation des expressions passées en arguments :
   `1.5 + 0.8` ⟶ `2.3`
   `3.4 * 1.25` ⟶ `4.25`
2. affectation des paramètres :
   `x = 2.3`
   `y = 4.25`
3. exécution du corps de la méthode :
   rien dans ce cas (corps réduit au simple `return`)
4. évaluation de la valeur de retour (expression derrière `return`)
   `(x + y) / 2.0` ⟶ `3.275`
5. replacement de l'expression de l'appel par la valeur retournée :
   `z = 3.275;`



Évaluation d'un appel de méthode (résumé)

L'évaluation de l'**appel**

$$f(arg1,\ arg2,\ \ldots,\ argN)$$

d'une méthode définie par

$$typeR\ f(type1\ x1,\ type2\ x2,\ \ldots,\ typeN\ xN)\ \{\ \ldots\ \}$$

s'effectue de la façon suivante :

1. les *expressions* $arg1$, $arg2$, ..., $argN$ passées en argument sont évaluées
2. les valeurs correspondantes sont **affectées** aux paramètres $x1$, $x2$, ..., xN de la méthode f (variables locales au corps de f)

   Concrètement, ces deux premières étapes reviennent à faire :

   $$x1 = arg1, x2 = arg2, \ldots, xN = argN$$
3. le programme correspondant au corps de la méthode f est exécuté
4. l'expression suivant la première commande `return` rencontrée est évaluée...
5. ...et retournée comme résultat de de l'appel :
   cette valeur remplace l'expression de l'appel, i.e. l'expression

   $$f(arg1,\ arg2,\ \ldots,\ argN)$$

Évaluation d'un appel de méthode (résumé)

L'évaluation de l'**appel** d'une méthode s'effectue de la façon suivante :

1. les *expressions* passées en argument sont évaluées
2. les valeurs correspondantes sont **affectées** aux paramètres de la méthode
3. le corps de la méthode est exécuté
4. l'expression suivant la première commande `return` rencontrée est évaluée...
5. ...et retournée comme résultat de de l'appel :
   cette valeur remplace l'expression de l'appel

Les étapes 1 et 2 n'ont pas lieu pour une méthode sans arguments.

Les étapes 4 et 5 n'ont pas lieu pour une méthode sans valeur de retour (`void`).

Appel : autre exemple

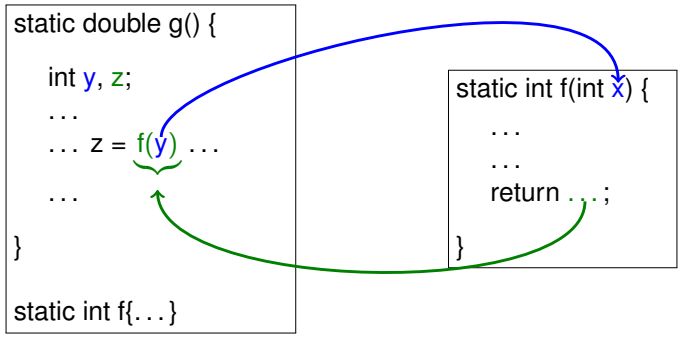
Une méthode peut appeler une autre méthode.

```
void afficheScore(int joueur, double points, double temps)
{
 System.out.println("  Joueur " + joueur
        + score(points, temps) + " points");
}

int score (double points, double tempsJeu)
{ // ... comme avant ... }
```

Appel : résumé

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



Résumé du jargon

« *Appeler la méthode* `f` » = utiliser la méthode `f` : `x = 2 * f(3);`

« *3 est passé en argument* » = (lors d'un appel) la valeur 3 est copiée dans un paramètre de la méthode :

`x = 2 * f(3);`

« *la méthode retourne la valeur de y* » = l'expression de l'appel de la méthode sera remplacée par la valeur retournée

```
 return y;
}
...
x = 2 * f(3);
```

Autres exemples : « `cos(0)` retourne le cosinus de 0 », « `cos(0)` retourne 1 ».

Le passage des arguments (1)

Considérons la situation suivante (pseudo-code) :

```
static 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 :

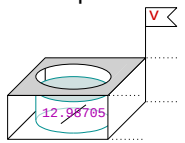
- L'argument `v` est **passé par valeur** si `methode` ne peut pas modifier `v1` : `v` est une **copie locale** de `v1`.
- L'argument `v` est **passé par référence** si `methode` peut modifier `v1`

Mais que veut dire «modifier `v`» ?

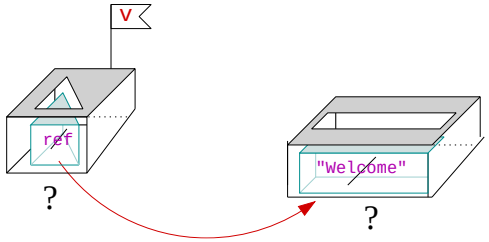
Le passage des arguments (2)

Java ne manipule pas les types élémentaires comme les types évolués :

Modifier `v` pour un type élémentaire n'a qu'une seule interprétation possible :



Pour un type évolué :



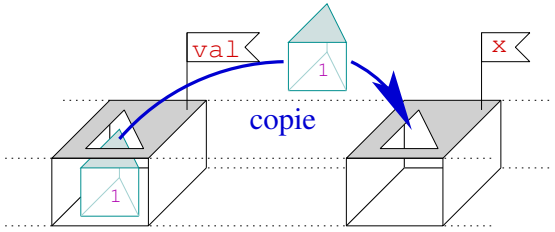
Le passage des arguments (2)

Il y a donc deux questions à poser au lieu d'une :

```
static void methode(Type v) { // Type :type EVOLUÉ
// traitement modifiant l'objet référencé par v
// traitement modifiant v lui même (référence)
}
// ailleurs:
Type v1 = ..; // initialisation de v1
methode(v1);
//1. v1 est-elle modifiée ici?
//2. l'objet référencé par v1 est-il modifié
//   ici?
```

Passage d'argument par valeur : schéma

En Java, il n'existe que le passage par valeur : une méthode travaille toujours sur une copie de la valeur qui lui est passée en paramètre



Passage par valeur : type élémentaire

```
static 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???
```

☞ Si `Type` est un type élémentaire
la réponse à la question dans le code est NON !!

Passage par valeur : type évolué

```
static void methode(Type v) { // Type :type EVOLUÉ
// traitement modifiant l'objet référencé par v
// traitement modifiant v lui même (référence)
}
// ailleurs:
Type v1 = ..; // initialisation de v1
methode(v1);
//1. v1 est-elle modifiée ici?
//2. l'objet référencé par v1 est-il modifié
//   ici?
```

- On a toujours un passage par valeur donc 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`** !

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

Exemple de passage par valeur (type élémentaire)

```
public static void main(String[] args) {
 int val = 1;
 m(val);
 System.out.println(" val=" + val);
}

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

L'exécution de ce programme produit l'affichage :
`x=2 val=1`

Ce qui montre que les modifications effectuées à l'intérieur de la méthode `m()` **ne** se répercutent **pas** sur la variable extérieure `val` associée au paramètre `x` et passée par valeur.

Type évolué : modification de la référence

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

L'exécution de ce programme produit l'affichage :
`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 !

Type évolué : modification de l'objet référencé

```
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
               // référencé par x
   System.out.print("x[0]= " + x[0]);
}
```

L'exécution de ce programme produit l'affichage :
`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.)

Entête

Toute méthode est caractérisée par un **entête**

- nom
- paramètres
- type de (la valeur de) retour

Syntaxe : $type\ nom\ (\ \overbrace{type_1\ id\_param_1\ ,\ \ldots,\ type_N\ id\_param_N}^{liste\ de\ paramètres}\ )$

Au niveau de ce cours, on ajoutera le mot clé `static` au début de chaque entête.
Mais deviendra une exception dans le cours « Programmation Orientée Objet ».

Exemples d'entêtes :

```
static double moyenne(double x, double y)

static int nbAuHasard()
```

Entête – Bonnes pratiques

- Une méthode ne doit faire que ce pour quoi elle est prévue
  Ne pas faire des choses cachées («*effets de bords*») ni modifier de variables extérieures (non passées comme arguments)
- Choisissez des **noms pertinents** pour vos méthodes et vos paramètres

  Cela augmente la lisibilité de votre code (et donc facilite sa maintenance).

  ☞ Il est en particulier très important que le nom représente bien ce que doit faire la méthode
- Commencez toujours par écrire l'entête de votre méthode :

  Demandez-vous ce qu'elle doit recevoir et retourner.

Définition des méthodes

La **définition** d'une méthode sert, comme son nom l'indique, à définir ce que fait la méthode :

- spécification du **corps** de la méthode

Syntaxe :

```
type nom ( liste de paramètres )
{
    instructions du corps de la méthode;
    return expression;
}
```

Exemple :

```
static double moyenne (double x, double y)
{
   return (x + y) / 2.0;
}
```

Corps de méthode

Le corps de la méthode est donc un **bloc**
dans lequel on peut utiliser les paramètres de la méthode (en plus des variables qui lui sont propres).

La valeur retournée par la méthode est indiquée par l'instruction :

`return` *`expression`*`;`

où l'*`expression`* a le même *type* que celui retourné par la méthode.

```
static double moyenne (double x, double y)
{
   return (x + y) / 2.0;
}
```

L'instruction `return` fait deux choses:

- elle précise la valeur qui sera fournie par la méthode en résultat
- elle met fin à l'exécution des instructions de la méthode.

L'expression après `return` est parfois réduite à une seule variable ou même à une valeur littérale, mais ce n'est pas une nécessité.

Remarques sur l'instruction `return` (1/4)

Il est possible de placer *plusieurs* instructions `return` dans une même méthode.

Par exemple, une méthode déterminant le maximum de deux valeurs peut s'écrire avec une instruction `return` :

```
static double max2(double a, double b)
{
  double m;
  if (a > b) {
    m = a;
  } else {
    m = b;
  }
  return m;
}
```

ou deux :

```
static double max2(double a, double b)
{
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Remarques sur l'instruction `return` (2/4)

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

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

Remarques sur l'instruction `return` (3/4)

`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
}
```

Remarques sur l'instruction `return` (4/4)

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 : pas de return si n <= 0 !
}
```

```
static double lire() {
   Scanner keyb = new Scanner(System.in);
   double n = 0.0;
   do {
       System.out.print("Entrez un nombre strictement positif : ");
       n = keyb.nextDouble();
   } while (n <= 0.0);
   return n;
}
```

Méthodes sans valeur de retour

Quand une méthode ne doit fournir aucun résultat (on appelle de telles méthodes des « **procédures** ») :

☞ définir une méthode **sans valeur de retour**

On utilise alors le type particulier `void` comme type de retour.

Dans ce cas la commande de retour `return` est optionnelle :

- soit on ne place aucun `return` dans le corps de la méthode
- soit on utilise l'instruction `return` sans la faire suivre d'une expression: `return;`

Exemple :

```
static void afficheRacine(double a)
{
  if (a < 0.0) {
    return; /* Grâce à ce return, on quitte la fonction avant *
             * de calculer sqrt(a) si a est négatif.          */
  }
  System.out.println(Math.sqrt(a));
  // il n'est pas nécessaire de mettre un return ici
}
```

Méthodes sans paramètre

Il est aussi possible de définir des méthodes **sans paramètre**.
Il suffit, dans l'entête, d'utiliser une liste de paramètres vide : `()`

Exemple :

```
private static Scanner clavier = new Scanner(System.in);

public static void main(String[] args)
{
  int val = saisieEntier();
  System.out.println(val);
}

static int saisieEntier()
{
  int i;
  System.out.println("entrez un entier: ");
  i = clavier.nextInt();
  return i;
}
```

La méthode `main`

`main` est aussi une méthode avec un nom et un entête imposés.

Par convention, tout programme Java doit avoir une méthode `main`, qui est appelée automatiquement quand on exécute le programme.

L'entête autorisée pour `main` est :

```
public static void main(String[] args)
```

Pour résumer : Méthodologie pour construire une méthode

1. clairement identifier ce que **doit faire** la méthode
   (ce point n'est en fait que conceptuel, on n'écrit aucun code ici !)
   - ☞ ne pas se préoccuper ici du *comment*, mais bel et bien du **quoi** !

     Les instructions dans le corps de la méthode dont la finalité n'est pas le calcul de la valeur de retour, ou qui modifient des objets extérieurs à la méthode (non passés en paramètre), sont appelées des « **effets de bord** ».
2. quels **arguments** ?
   - ☞ que doit recevoir la méthode pour faire ce qu'elle doit ?

Pour résumer : Méthodologie pour construire une méthode

3. quel type de retour ?
   - ☞ que doit « retourner » la méthode ?
     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`
4. (maintenant, et seulement maintenant) Se préoccuper du *comment* :
   comment faire ce que doit faire la méthode ?
   - ☞ c'est-à-dire écrire le corps de la méthode

La surcharge de méthodes

En Java , il est de ce fait possible de définir **plusieurs méthodes de même nom** si ces méthodes n'ont pas les mêmes listes de paramètres : nombre ou types de paramètres différents.

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

La surcharge de méthodes : 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("couple : " + x1 + "," +  x2);
}
```

`affiche(1)`, `affiche(1.0)` et `affiche(1,1)` produisent alors des affichages différents.