Chapitre 2: L'essentiel pour commencer

(NFA031 - Jour)

V. Aponte

Cnam

Octobre 2012

1. Èléments fondamentaux.

Aspects fondamentaux de la programmation

Programmes = Données + Instructions

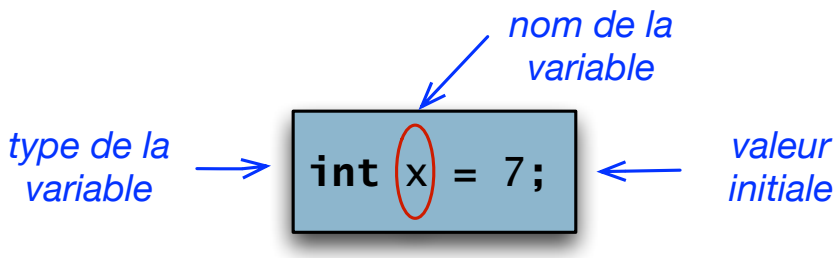
- Données $\Rightarrow$ comprendre variables + types.
- Instructions $\Rightarrow$ comprendre structures de contrôle + sous-programmes.

Aspects fondamentaux de la programmation (2)

Ecrire des programmes $\Rightarrow$ comprendre :

- syntaxe des instructions :
  $\Rightarrow$ *comment agencer mot-clés et symboles*.
- sémantique des instructions :
  $\Rightarrow$ *comportement à l'exécution*.

Variables et types : un aperçu



Variables : C'est quoi ? (sémantique) :

- un nom (ici x), associé à un emplacement de la mémoire,
- qui contient une donnée (ici 7),
- et qui possède type (ici `int`) ;

et un type ? décrit la nature d'une donnée. Le type `int` signifie «nombre entier».

Instructions : un aperçu

Ex : une instruction «d'affectation»

```
x = 10 + 3;
```

sémantique Ordre à exécuter.

Ici : «*calculer 10+3, puis, enregistrer le résultat dans l'emplacement associé à la variable x*».

structure de contrôle : un aperçu

Ex : une boucle

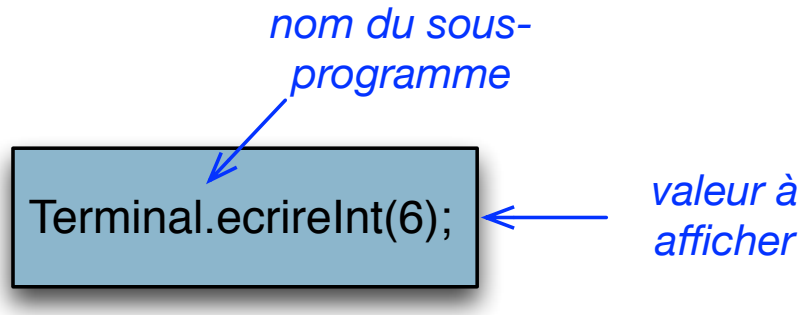
```
for (int i=0; i<=10; i++){
        Terminal.ecrireIntln(i);
     }
```

sémantique Instruction qui change le «flux d'exécution», c.a.d., le fait que les instruction du programme s'exécutent séquentiellement.

Ex : la boucle «for» permet de «répeter» des instructions.

boucle Instruction de répétition.

Sous-programmes : un aperçu



- le nom du sous-programme (ici : `Terminal.ecrireInt`).
- correspond à un regroupement d'instructions à exécuter(ici, afficher le nombre entier x qui lui est passé en argument),
- Quand ce nom apparaît ses instructions associées sont exécutées.
- Dans tous les langages : «bibliothèques» avec beaucoup de sous-programmes utilitaires.

2. Les programmes en Java.

Un premier programme

Conversion d'une somme en euros, lue au clavier, vers son équivalent en francs :

```
public class Conversion {
 public static void main (String[] args) {
    double euros, francs;
    Terminal.ecrireStringln("Somme en euros? ");
    euros = Terminal.lireDouble();
    francs = euros * 6.559;
    Terminal.ecrireStringln("Conversion= "+ francs);
   }
}
```

Structure d'un programme Java

```
public class <nom-du-programme> {

    <déclaration-optionelle-variables-et-sousprogrammes>

    public static void main (String[] args) {

        <déclarations-et-instructions>
    }
    <déclaration-optionelle-variables-et-sousprogrammes>
}
```

Structure d'un programme Java

Un programme Java est composé, au minimum de :

- Une «classe principale» :

  public class `<nom-du-programme>`

- contenant un sous-programme ou «méthode principale» :

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

- contenus classe principale et méthode main : délimités entre accolades `{ }` .

- **public, class, static, void, main** $\Rightarrow$ mots réservés : on ne peut pas les employer pour nommer variables, programmes, etc.

Structure d'un programme Java (2)

Nom programme : nom de la classe principale =

- nom programme en entier,
- nom du fichier contenant le programme + extension `.java`.

Ex : le programme `Conversion` doit se trouver dans le fichier `Conversion.java`

méthode `main` : obligatoire dans tout programme Java.
Exécuter `Conversion`, c'est exécuter sa méthode `main`.

déclarations-instructions : description des variables et instructions à exécuter par `main`.

Comprendre `Conversion`

```
1 public class Conversion {
2  public static void main (String[] args) {
3    double euros, francs;
4    Terminal.ecrireStringln("Somme en euros? ");
5    euros = Terminal.lireDouble();
6    francs = euros * 6.559;
7    Terminal.ecrireStringln("Conversion= "+ francs);
8   }
9 }
```

- ligne 3 : Déclaration des variables : `euros`, et `francs` de type `double` (type des nombre réels), puis
- ligne 4 : afficher le message “Somme en euros ?”, puis
- ligne 5 : saisie d’un nombre, puis affectation dans `euros` ; puis
- ligne 6 : calcul de la conversion et affectation du résultat dans `francs`, puis
- ligne 7 : afficher un message avec le résultat.

Les identificateurs (noms)

Pour nommer les variables, les programmes, les sous-programmes.

Identificateur : mot qui commence par une lettre, ou par _, ou par `$` , suivi ou non de caractères parmi : a . . . z, A . . . Z, 0..9, _, `$`, et caractères *Unicode*.

En plus, ils doivent respecter :

- Ne peut pas être un mot réservé : (`abstract, boolean, if, public, class, private, static`, etc).
- Caractères interdits : (`^, [, ], {, +, -`, . . .).

Exemples : `a`, `id_a` et `X1` sont des noms de variables valides, alors que `1X` et `X-X` ne le sont pas.

3. Types de données en Java.

Les types

Type Description de la nature d'une donnée. Ex : un nombre entier (`int`), un nombre réel (`double`), un tableau, un booléen (`boolean`).

Constantes A chaque type correspond un ensemble de valeurs constantes de ce type. Ex : pour le type `int`, les constantes sont 1, 2, -4, etc.

Opérations Chaque type possède un ensemble d'**opérateurs** pour manipuler ses valeurs. Ex : +,-,*,/ sont des opérateurs sur les types `int` et `double`.

Les types en Java

- Types primitifs : ce sont les types des données élémentaires (une seule donnée à la fois :un entier, un booléan, etc).
  - en Java ils sont tous prédéfinis ;
  - nous utiliserons : `int`, `double`, `boolean`, `char` (il y en a d'autres).
- Types référence : ce sont les types des objets.
  - les objets regroupent en général plusieurs données (un objet n'est pas une donnée élémentaire)
  - certains sont prédéfinis : nous utiliserons `String`
  - mais on peut définir ses propres types objet.

Quelques types prédéfinis en Java (int, double)

`int` nombres entiers représentables en machine sur 32 bits (31 plus le signe) $\{-2^{31}, \ldots, 2^{31}\}$.
Constantes : `-7, 0, 1, 2....`
Opérateurs : +,-,*,/, `%`, ==, >, <, >=, !=
Attention : division entière si tous les opérandes sont des entiers !

```
/* Exemples  */
   int x=2; int y = 6;
   x = x+y*3;
```

`double` les réels (à précision double, 64 bits).
Constantes : `0.0, 0.1, ...18.58 ...`
Mêmes opérateurs, sauf modulo.

Quelques types en Java (boolean)

`boolean` modélise les deux valeurs de vérité en logique.
Constantes : `true` et `false`.
Opérateurs : `&&` (et), `!` (négation), `||` (ou).

```
/* Exemples    */
   boolean a = false;
   boolean b;
   b = true || false;      // b ← true
   b = true && false;      // b ← false
   b = !a;                 // b ← true
```

Type `char`

`char` caractères Unicode (sur 16 bits). Constantes : les caractères ASCII entourés de guillemets simples, p.e : `'a'`, `'2'`, `'@'` ; Opérateurs : de comparaison (=, <, >, !=,...)

```
/* Exemples    */
   char a = 'c';
   Terminal.ecrireChar(a);        // affiche c
   Terminal.ecrireChar('a');      // affiche a
   Terminal.ecrireChar('\t');     // affiche une tabulation
```

Type `String`

`String` chaînes de caractères.

Constantes suites de caractères entourées de guillemets doubles :
`"entrer une somme en euros :?","coucou", "a"`
..., y compris la chaîne vide `""`

Opérateurs + est l'opérateur de concaténation.

```
/* Exemples    */
   int x = 3;
   Terminal.ecrireString("a");           // affiche a
   Terminal.ecrireString("a" + "b");     // affiche ab
   Terminal.ecrireString("x= " + x);     // affiche x = 3
```

Opérateurs de comparaison

Comparent deux *expressions* **de même type** (primitif).
Le résultat est un booléan.

==	egalité
<	plus petit
>	plus grand
>=	plus grand ou égal
<=	plus petit ou égal
!=	(différent)

Les expressions

Expression

- Soit une valeur constante, soit une variable, soit une opération composée d'opérateurs et opérandes.
  - opérandes : valeurs constantes, variables, ou appels de fonctions.
- valeur de l'expression : résultat d'appliquer les opérateurs sur les opérandes. On dit aussi évaluer.

Expression	Valeur calculée	Particularités
7	7	expression constante
7 + 6.5	13.5	expression arithmétique
"x" + "y"	"xy"	expression de concaténation sur des chaînes.
x + 3	?	dépend de la valeur de x en mémoire
Math.min(3,4)	3	appel à une fonction
Terminal.lireInt()	valeur saisie	appel à une fonction

Exemples d'expressions

Supposons qu'en mémoire : x=3, a=true, b =false, c='x'.

Expression	Valeur
7 > 4	true
7 >= 7	true
7 != 7	false
'a' < 'b'	true
c > 'c'	true
c == 'c'	false
!a	false
a && !b	true
x < 10	true
(x > 10) == ('b' == c)	true

Exemples d'instructions (et non pas d'expressions) :

- `int x = 1;`
- `System.out.println(5);`

Où trouver les expressions ?

Un peu partout ! Essentiellement :

- membre droit d'une affectation ou initialisation,
- arguments d'un appel à un sous-programme.

```
/* Expressions dans toutes les lignes de ce code!  */
   int x = 1;
   int y = x+2;
   boolean a = false;
   b = true || false;
   if (y>x)
     x = Terminal.lireInt();
   else
     y = Math.min(x*2,y);
```

Opérateurs d'incrément et décrement

Applicables sur tous les types numériques et sur `char`

```
i++;    équivaut à i=i+1;
i--;    équivaut à i=i-1;
```

Opérateurs d'affectation :

```
x -= y;    équivaut à    x=x-y;
x += y;    équivaut à    x=x+y;   (aussi avec String)
x *= y;    équivaut à    x=x*y;
x /= y;    équivaut à    x=x/y;
```

Opérateurs conditionnel

Permet d'obtenir une valeur résultat au moyen d'une conditionnelle.

```
<expr-bool> ? <expression1> : <expression2>;
```

- si <*expr-bool*> est vrai, renvoie <*expression1*>,
- sinon renvoie <*expression2*>

```
anneeSuivante = (mois == 12) ? (annee+1) : annee;
```

`anneeSuivante` est `(annee+1)` si on est en décembre,
égale à `annee` sinon.

Précédence des opérateurs

Les expressions composées de plusieurs opérateurs sont évaluées de la gauche vers la droite, selon des *règles de précédence* indiquant la priorité des opérateurs les uns par rapport aux autres.

```
2 + 3 * 4      équivaut à    2 +(3*4) = 14
               et non pas à             20

2 + 3 > 7      s'évalue en   true

2 + (3 > 7)    produit une erreur
```

Précédence des opérateurs

!	négation
* / %	multiplication, division, modulo
+ −	addition, soustraction
> < >= <= == !=	comparaison
&& ||	booléans
=	affectation

4. Variables et affectation.

Déclaration de variables

Avant d'utiliser une variable on doit la déclarer.

Syntaxe

```
<type-de-la-variable> <nom-variable>;
<type-de-la-variable> <nom-variable> = <expression>
```

```
double euros; char reponse = 'n';
int x = 0;
```

- type de la variable, puis son nom (*identificateur*),
- optionnellement, on donne une valeur initiale,
- les types de la valeur et de la variable doivent correspondre.

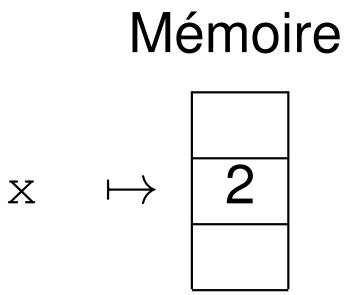
Déclaration de variables (sémantique)

Exemple :

```
int x = 2;
```

Sémantique :

- on réserve (mémoire) un emplacement suffisant pour stocker une valeur du type de la variable (ici 32 bits),
- on y met la valeur (si elle est donnée). Elle doit être du bon type.



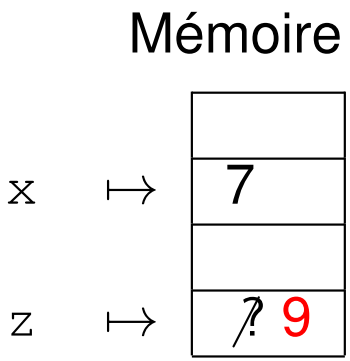
Utiliser une variable

Equivaut à *utiliser la valeur qu'elle contient en mémoire*.

Elle doit être **déclarée** et **initialisée**.

```
int x = 7; int z;
z = x + 2;
```

Calculer `x+2` $\Rightarrow$ chercher en mémoire la valeur de x.
`x+2` $\Rightarrow$ 7+2 $\Rightarrow$ 9.



Instruction d'affectation

Syntaxe

```
<nom-variable> = <expression>;
```

```
int x = 3;      // declarations
int y = 7;
x = x + y;      // affectation
```

A la compilation : les types à droite et à gauche du = doivent être compatibles.

Exécutée en 2 temps

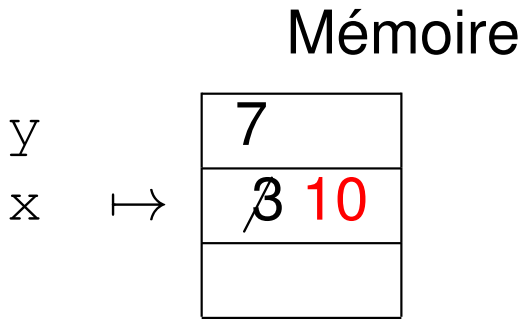
(1) calculer valeur à droite du = ;
(2) copier résultat dans (emplacement) pour variable à gauche.

Exemple : exécution affectation (2 temps)

Exécuter `x = x+y`

1. Evaluer expression **a droite** du symbole $=$
   $\Rightarrow \mathtt{x} + \mathtt{y} \Rightarrow 3 + 7 \Rightarrow 10$

2. Enregistrer résultat dans emplacement mémoire de la variable affectée :



Exercice pour tester votre compréhension

Ecrire un programme Java qui :

- échange les valeurs de deux variables x (initialisée avec 2) et y (initialisée avec 10), toutes eux de type entier.
- Votre programme doit afficher les valeurs de ces deux variables avant et après l"échange.
- dessinez les changements dans la mémoire pour toutes les variables du programme.

5. Les sous-programmes en Java (méthodes).

Sous-programmes et méthodes

```
1 public class Conversion {
2  public static void main (String[] args) {
3     double euros, francs;
4     Terminal.ecrireStringln("Somme en euros? ");
5     euros = Terminal.lireDouble();
6     francs = euros * 6.559;
7     Terminal.ecrireStringln("Conversion= "+ francs);
8    }
9 }
```

dans lignes 4, 5, 7 : **appels au sous-programmes** :

```
Terminal.ecrireStringln
```

Son rôle : afficher le message (de type **String**) qui lui est passé en argument.

Sous-programmes = méthodes

```
1 public class Conversion {
2  public static void main (String[] args) {
3    double euros, francs;
4    Terminal.ecrireStringln("Somme en euros? ");
5    euros = Terminal.lireDouble();
6    francs = euros * 6.559;
7    Terminal.ecrireStringln("Conversion= "+ francs);
8   }
9 }
```

- ligne 5 : sous-programme `Terminal.lireDouble`
  Son rôle est de «lire» un entier au clavier.
- En Java, les sous-programmes s'appellent «méthodes».

Sous-programmes = méthodes

Sous-programme (méthode)

Suite d'instructions regroupée sous un nom donné. Ses instructions sont exécutées à chaque appel (chaque fois que ce nom apparaît).

- il est invoqué, ou appelé (pour exécuter ses instructions),
- on peut lui fournir des entrées (arguments, paramètres),
- il peut retourner une valeur en résultat, qu'on pourra utiliser après son exécution.
- 2 sortes :
  - fonctions : retournent un résultat.
  - procédure : ne retournent pas de résultat.

Exemples d'appels de méthodes

- `Terminal.ecrireStringln("Bonjour")`
  - son argument : message "Bonjour" (type `String`),
  - son comportement : afficher `Bonjour`
- `Terminal.lireDouble()`
  - arguments : aucun
  - comportement : retourner la prochaine valeur de type `double` entrée au clavier.
- `System.out.print(5)` :
  - argument : 5
  - comportement : afficher son argument.

Comprendre les appels de méthodes

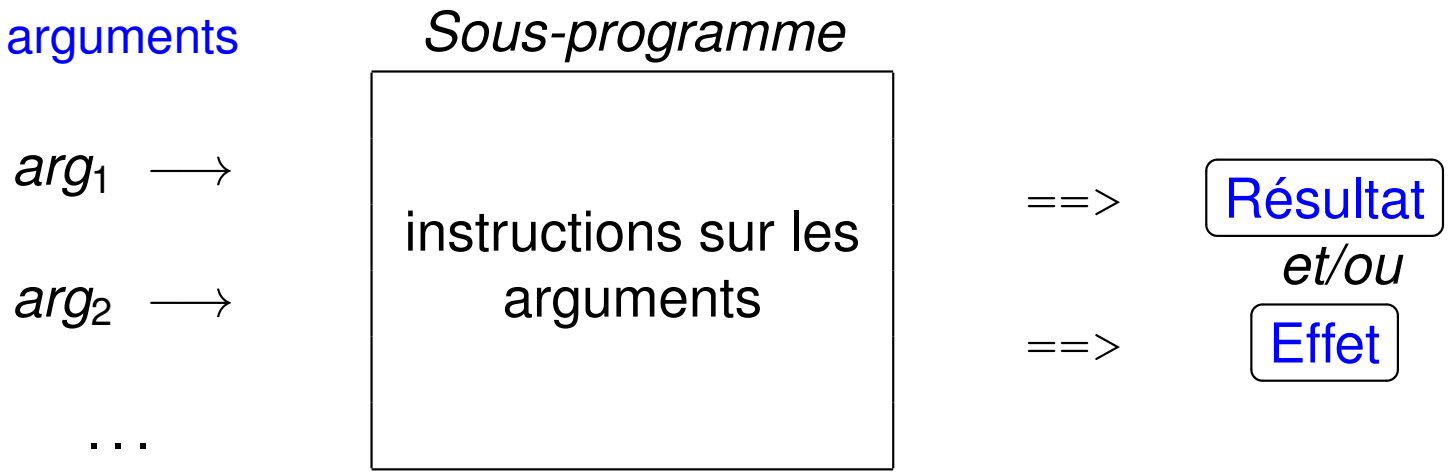
Syntaxe :

`<nom-sous-programme>` ($arg_1, arg_2, \ldots, arg_n$)

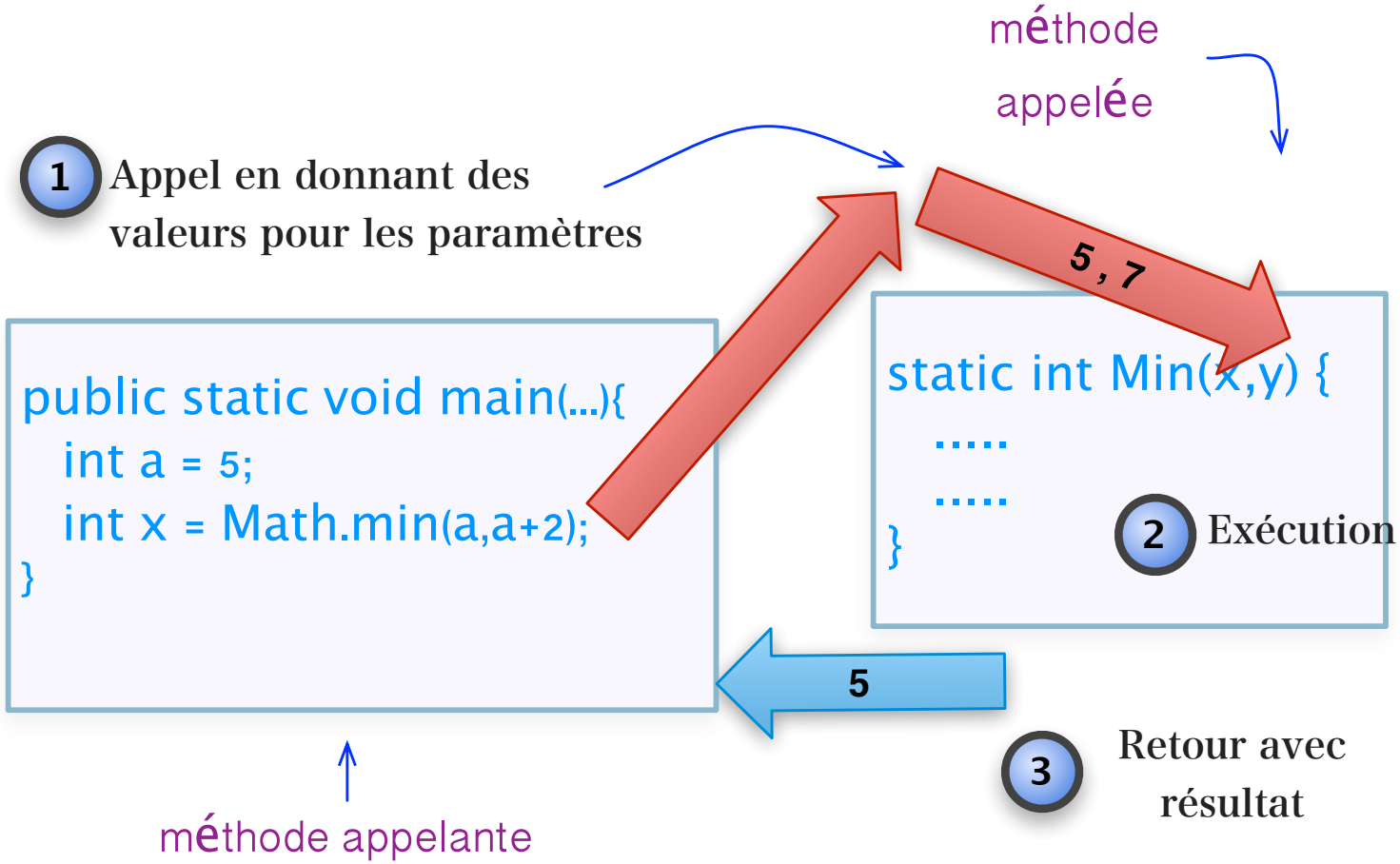
Sémantique :

- $arg_1, arg_2, \ldots, arg_n$: les entrées (appelés «arguments» ou «paramètres»). S'il n'y en a pas, on écrit ()
- le sous-programme est une «boîte noire» capable de réaliser des traitements sur ses arguments.
- il produit, soit une «valeur de retour», (ex : un entier), soit un «effet» (ex : un affichage), et pas de valeur de retour.

Comprendre les appels de méthodes



Exemple d'un appel



En mémoire, au retour de l'appel : $x \leftarrow 5$

Appels imbriqués

Appel dont un ou plusieurs arguments correspondent à d'autres appels de méthodes.

```
int x = 4;
int y = 10;
Terminal.ecrireInt(Math.min(x,y));
```

L'appel à `Terminal.ecrireInt` a pour argument un appel à `Math.min` (méthode qui retourne le plus petit parmi ses arguments).

Comprendre les appels imbriqués

Règle : Avant d'exécuter un appel, on doit calculer la valeur de tous ses arguments.

```
int x = 4;
int y = 10;
Terminal.ecrireInt(Math.min(x,y));
```

Conclusion : dans un appel imbriqué, on commence par exécuter l'appel le plus interne.

Comprendre les appels imbriqués

```
int x = 4;
int y = 10;
Terminal.ecrireInt(Math.min(x,y));
```

Exécution :

1. Calcul des arguments de `Math.min` : on récupère leurs valeurs en mémoire $\Rightarrow$ 4,5
2. Appel à `Math.min(4,5)` $\Rightarrow$ retourne 4
3. 4 est l'argument pour l'appel à `Terminal.ecrireInt`
4. Appel à `Terminal.ecrireInt(4)` $\Rightarrow$ affichage de 4

Écrire les appels des méthodes

L'écriture diffère selon que la méthode retourne ou non une valeur

méthode avec valeur résultat $\Rightarrow$ le résultat doit être récupéré : dans une variable, dans une expression, pour l'afficher, etc.

```
euros= Terminal.lireDouble();  Resultat mis dans euros

                    Resultat de Math.min affiche par Terminal.ecrireInt
Terminal.ecrireInt(Math.min(2,7));

x= Terminal.lireInt() + 3;      Additionne à 3 puis mis dans x
```

Comment écrire les appels des méthodes

méthode sans résultat $\Rightarrow$ Pas de valeur à récupérer. L'appel est une instruction à lui tout seul.

```
Terminal.ecrireStringln("Bonjour");
```

Exemples d'appels

Quelles sont les lignes erronées ? Qu'affichent les lignes correctes ?

```
int x;  int y = 10;
x = Terminal.lireInt();
Terminal.lireInt();
Terminal.ecrireInt(Terminal.lireInt());
Terminal.ecrireInt(Terminal.lireInt() + 4);
x = Math.min(Terminal.lireInt(), y);
Math.min(Terminal.lireInt(), y);
Terminal.ecrireInt(Math.min(Terminal.lireInt(), y));
y = Terminal.ecrireInt(5);
Terminal.ecrireInt(5);
```

6. Entrées/sorties standard (clavier/écran).

Les actions d'entrées/sorties

entrées **lire** des valeurs (saisies au clavier, ou à partir d'un fichier des données). Retournent la valeur lue.

sorties **écrire** des valeurs (à l'écran, ou dans un fichier de données). Ne retournent aucune de valeur.

Méthodes d'entrées/sorties en Java

`System` : classe prédéfinie dans la bibliothèque Java avec méthodes d'entrées/sorties.

`Terminal` : classe écrite par les enseignants, qui regroupe les fonctions de saisie/affichage sur le clavier/écran (pas de lecture/écriture sur les fichiers), pour tous les types primitifs utilisés dans ce cours.

Le fichier source `Terminal.java`, doit se trouver dans le même répertoire que vos programmes.

Affichage avec System

- `System.out.print` : affiche une valeur primitive ou un message qui lui est passé en paramètre.
- pas de valeur résultat ⇒ l'appel est une instruction

```
System.out.print(5);                    5

System.out.print(bonjour);      contenu de bonjour

System.out.print("bonjour");          bonjour

System.out.print("bonjour␣"+ 5 ); bonjour 5

System.out.print(5 + 2);                7
```

Affichage avec Terminal

- `Terminal.ecrireType(v);` où `Type` est un type parmi `int double`, `boolean`, `char` et `String`. Affiche la valeur `v` qui doit être du type indiqué par `Type`
- `Terminal.ecrireTypeln(v);` idem, plus un saut à la ligne.
- Pas de valeur résultat $\Rightarrow$ l'appel est une instruction,

```
Terminal.ecrireInt(5);
Terminal.ecrireChar('a');
Terminal.ecrireIntln(5);
Terminal.ecrireDoubleln(1.3);
```

Affiche :

```
5a5
1.3
```

Méthodes d'affichage dans Terminal

- `Terminal.ecrireInt(n);`
- `Terminal.ecrireDouble(n);`
- `Terminal.ecrireBoolean(n);`
- `Terminal.ecrireChar(n);`
- `Terminal.ecrireString(n);`
- `Terminal.ecrireIntln(n);`
- `Terminal.ecrireDoubleln(n)` ...

Affichage avec le bon type

On doit donner une valeur du bon type.

```
Terminal.ecrireString(5 + 2);
                ---> Erreur de Typage!

Terminal.ecrireString("bonjour " + 5 + 2 );
                -> affiche: bonjour 52

Terminal.ecrireString("bonjour " + (5 + 2) );
                -> affiche: bonjour 7
```

Saisie avec Terminal

- `Terminal.lireType();` où `Type` est l'un parmi `int` `double`, `boolean`, `char` et `String`.
- La saisie se fait lors de la validation par un retour chariot.
- Si la valeur saisie n'est pas du type `Type`, produit une **erreur à l'exécution**, qui arrête l'exécution du programme.
- Si la valeur saisie est du bon type, **retourne cette valeur en résultat** $\Rightarrow$ l'appel est une expression (et doit récupérer le résultat).

Exemples de saisie avec Terminal

```
int x;
double y;
char c;
x = Terminal.lireInt() + 4;
y = Terminal.lireDouble();
c = Terminal.lireChar();
Terminal.ecrireInt(Terminal.lireInt());
```

Méthodes de saisie dans Terminal

- `Terminal.lireInt()`
- `Terminal.lireDouble()`
- `Terminal.lireBoolean()`
- `Terminal.lireChar()`
- `Terminal.lireString()`

Méthodes et classes

- Les méthodes que l'on peut appeler dans un programme sont toujours **définies dans une classe**.
- Certaines méthodes sont **statiques** : on les invoque avec la syntaxe

```
<classe-de-définition>.<méthode>(<arguments>);
```

  - on fait précéder le nom de la méthode (p.e : `lireDouble`),
  - par le nom de la classe où elle est définie (p.e : `Terminal`) suivie d'un «.»
  - et on donne les arguments entre parenthèses.

Exemple : `Terminal.ecrireInt(5);`

Méthodes statiques et non statiques

- Les méthodes *statiques* sont déclarées avec le mot clef **static**. Ex : `main` est une méthode statique.
- elles sont toujours appelées avec le nom de leur classe. Ex : méthode `ecrireInt` de la classe `Terminal` : `Terminal.ecrireInt(5)`,
- Les méthodes non statiques sont appelées *sur des objets*.
- Dans nfa031, nou ne verrons que les méthodes statiques.

7. Conversions entre types.

Expressions qui mélangent des types : autorisées (dans certains cas)

Java autorise certaines expressions ou affectations qui *mélangent des valeurs et variables de types différents*.

- *focus sur types prédéfinis* : primitifs + `String` ;

Exemples valides :

- (expression) 3 + 4.3 = 7.8 (`int` + `double`) $\Rightarrow$ `double`
- (expression) 3+ "abc"= "3abc", (`int` + `String`) $\Rightarrow$ `String`
- (affectation) `double z = 5;` (une variable de type double reçoit une valeur de type int).

Opérateurs et types des opérandes (types prédéfinis)

- `-`, `*`, `%` $\Rightarrow$ applicables sur n'importe quel type primitif numérique.
- `+` $\Rightarrow$ applicable sur types numériques *et aussi* sur `String`.
- Selon le type des opérandes, l'opération réellement exécutée diffère.
  - `(+)` : on exécute une concaténation entre chaînes, ou une addition entre nombres ;
  - `(*)` : on execute une multiplication différente selon que les nombres multipliés ont ou non une partie décimale.
- En coulisses, les opérations réellement exécutées sont applicables uniquement sur des opérandes de même type.

Nécessaire

Mettre toutes les opérandes au même format avant d'effectuer les opérations.

Qu'est-ce qu'une conversion entre types ?

Conversion entre types

Changement du type d'une valeur, et en particulier de sa représentation en mémoire. Il y a deux sortes de conversions :

- Implicite : c'est le compilateur qui décide, sans que ce soit demandé par le programmeur. Uniquement dans certains cas.
  - `double m = 3;` Le type (int) de 3, et son format (32 bits) est changé en type double et format 64 bits (nombre à virgule) ;
  - le changement est opéré à l'exécution, *avant* d'éffectuer l'affectation en mémoire.
- Explicite : c'est le programmeur qui le demande, par une syntaxe dediée.

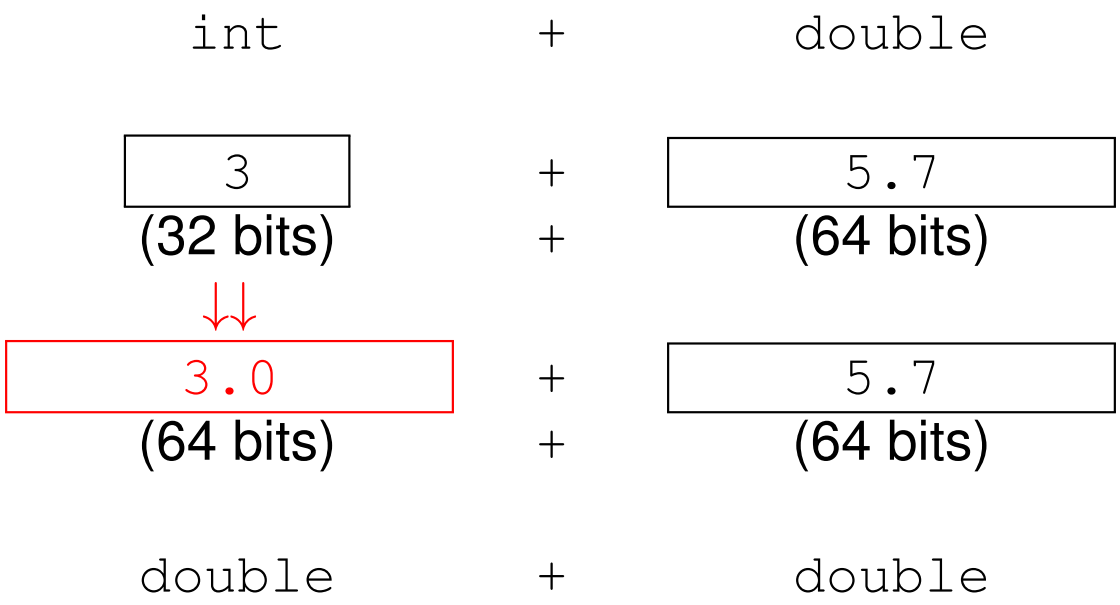
Attention : on ne modifie jamais *le type des variables*. Les conversion s'éffectuent toujours sur des valeurs.

Conversions implicites entre types (primitifs)

- si *opérandes de même type* :
  - pas besoin de conversion ;
  - le résultat est du type commun ;
  - ex : 3+4 = 7, (`int` + `int`) $\Rightarrow$ `int`
- si *opérandes des types différents* :
  - Le compilateur réalise une *conversion implicite* (automatique) des opérandes de manière à les rendre uniformes.
  - vers le type le «plus grand» si expression arithmétique ;
  - ou vers `String`, si concaténation avec `String`
  - si la conversion est impossible : erreur à la compilation.

Conversions implicites entre int et double

Le compilateur éffectue une conversion implicite vers un type commun (sans perte d'information).



Conversions implicites entre int et double

- implicite = automatique : le compilateur la réalise sans prévenir !
- seules les conversions sans perte d'information se font automatiquement :
  - si nécessaire, réalisée dans le sens : `int` $\rightarrow$ `double`.
  - jamais réalisée dans le sens : `double` $\rightarrow$ `int`. On perdait la partie décimale.

Exemple :
`3.5 (double)` ne peut pas être convertit **implicitement** vers un `int`.
On perdrait la partie décimale.

Conversions implicites entre String et primitifs

- réalisée uniquement si *concaténation avec opérande String* ;
- le résultat est un String.

Exemples :

- `6+2+"abc" = "62abc"` , (`int + int + String`) $\Rightarrow$ `String`
- `(6+2) +"abc" = "8abc"` ,
  ((`int + int`)+ `String`) = `int + String` $\Rightarrow$ `String`

Affectations (int, double) avec conversions

(type variable)	=	(type valeur)	conversion nécessaire ?
`int`	=	`int`	non
`double`	=	`double`	non
`double`	=	`int`	implicite : `int` → `double`
`int`	=	`double`	explicite (cast) sinon erreur

Exemple :

```
double m = 6;   // conversion int → double
int x = m;      // erreur
```

Division entière

La division de deux entiers donne en résultat un entier

```
5/2  donne en résultat  2
      et non pas  2.5
```

Pourquoi ?

Dans :

```
double m = 5/2;
```

quelle est la valeur de m ?

Conversions explicites (*Cast*)

La conversion entre types avec perte d'information est autorisée, lorsqu'elle est réalisée explicitement par le programmeur.

Par exemple, pour convertir `3.67` vers un `int`, il suffit de d'écrire :

$$\texttt{(int) 3.67} \Rightarrow 3$$

ce qui donne la valeur `3` de type `int`. Une conversion explicite, appelée *cast* en Java, prend la forme `(type_cible) v`, où `type_cible` est le type vers lequel on souhaite faire la conversion.

Attention : cette conversion n'est possible que si elle a un sens vis-à-vis des types. Ainsi, par exemple, la conversion
`(boolean) 5` est invalide car `boolean` n'est pas un type numérique.

Conversions explicites pour `char`

Conversions explicites entre types numériques et `char` :

- `(char) 97` $\Rightarrow$ `'a'`
- `(int) '+'` $\Rightarrow$ `43`

Règles de style en Java

- Java n'impose pas des règles de formattage des programmes.
- On peut écrire tout un programme sur une seule ligne : c'est un entrée valide pour le compilateur. Mais ce n'est pas très lisible pour les humains.
- Certaines conventions d'écriture sont devenues des «standards» : elles visent à améliorer la lisibilité des programmes par les programmeurs eux-mêmes.

Règles de style (2)

- Les noms des classes débutent par une majuscule (`Terminal`, `Conversion`).
- Les noms des variables et méthodes débutent par une minuscule (`main`, `x`, `euros`, `lireInt`).
- Les noms composés se font par adjonction de plusieurs mots, chaque mot débutant par une majuscule (`CompteBancaire`, `lireInt`).
- Chaque variable initialisée est déclarée (toute seule) sur une ligne.

Règles de style (3)

- Les variable essentielles au programme sont déclarées en début de la méthode main. Les variables auxiliaires, juste au moment où elles sont nécessaires.

- Chaque nouvelle structure est décalée de 2 ou 3 caractères à droite par rapport à la structure qui la contient. On parle **d'indentation**.

```
public class Conversion {
  public static void main (String[] args) {
```

Règles de style (4)

- Les instructions son indentées à droite par rapport au bloc qui les contient.

```
public static void main (String[] args) {
   double euros;
   double francs;
   Terminal.ecrireStringln("Somme␣en␣euros?␣");
   euros = Terminal.lireDouble();
   ...
```

- Chaque instruction est écrite sur une ligne. Toutes les instructions d'un bloc sont alignées sur la même colonne.

Règles de style (5)

- L'accolade fermante d'un bloc est alignée sur la même colonne que le début de la structure qu'elle délimite.

```
public class Conversion {
  public static void main (String[] args) {
    ....
    francs = euros * 6.559;
    Terminal.ecrireStringln("Conversion= "+ francs);
  }
}
```