Mise à niveau en Java

Cours 0

Stéphane Airiau

Université Paris-Dauphine

Ces transparents

➪ **Eléments de base**

ou bien : (presque) tout ce dont vous avez besoin pour ré-écrire les algorithmes vus en cours d'algorithmique **sans** utiliser des objets !

- Variables, opérateurs, type d'une expression
- Tests, boucles
- Tableaux
- méthodes

Instructions et commentaires

Il est important de bien commenter son code !
Deux possibilités pour commenter :

```
1 // la suite est un commentaire
```

```
1 /* ceci est un commentaire
2  sur plusieurs
3  lignes */
```

On verra plus tard qu'on pourra formater les commentaires pour générer automatiquement de la documentation (voir chapitre 10 des notes de cours et `javadoc`).

Ce qui n'est pas un commentaire devra être compris par `Java`
⇨respecter la grammaire de `Java`.

Une instruction se termine toujours par un point virgule `;`

Types élémentaires

type élémentaire	nombre de bits	interval de valeurs
`boolean`	1	deux valeurs `true` et `false`
`byte`	8	un entier entre -128 et 127
`short`	16	un entier entre -2^{15}=−32768 et $2^{15}-1$=32767
`int`	32	un entier entre $-2^{31} \approx -2.1 \cdot 10^9$ et $2^{31}-1 \approx 2.1 \cdot 10^9$
`long`	64	un entier entre $-2^{63} \approx -9.2 \cdot 10^{18}$ et $2^{63}-1 \approx 9.2 \cdot 10^{18}$
`char`	16	caractère unicode, il y a 65536 codes
`float`	32	nombre flottant norme IEEE
`double`	64	nombre flottant norme IEEE

Variables : déclaration et initialisation

- déclaration simple :
  `<type> <nom>;`
- déclaration avec affectation :
  `<type> <nom> = <valeur dans le type> | <variable> | <expression>;`
- déclaration multiple :
  `<type> <nom`$_1$`>, <nom`$_2$`>, ..., <nom`$_k$`>;`
- déclaration multiple avec affection partielle :
  `<type> <nom`$_1$`>, <nom`$_2$`>= <valeur dans le type>, ..., <nom`$_k$`>;`

Exemples

```
1 short population;
2 population = 30000;
```

Exemples

```
1 short population;
2 population = 30000;
```

```
1 short population = 1.000.000;
```

Exemples

```
1 short population;
2 population = 30000;
```

```
1 short population = 1.000.000;
```

Exemples

```
1 short population;
2 population = 30000;
```

```
1 short population = 1.000.000;
```

```
1 long nbParticules = 10.000.000.000;
```

Exemples

```
1 short population;
2 population = 30000;
```

```
1 short population = 1.000.000;
```

```
1 long nbParticules = 10.000.000.000;
```

Exemples

```
1 short population;
2 population = 30000;
```

```
1 short population = 1.000.000;
```

```
1 long nbParticules = 10.000.000.000;
```

```
1 long nbParticules = 10.000.000.000L;
```

Exemples

```
1 short population;
2 population = 30000;
```

```
1 short population = 1.000.000;
```

```
1 long nbParticules = 10.000.000.000;
```

```
1 long nbParticules = 10.000.000.000L;
```

```
1 char lettre = 'c';
```

Exemples

```
1 short population;
2 population = 30000;
```

```
1 short population = 1.000.000;
```

```
1 long nbParticules = 10.000.000.000;
```

```
1 long nbParticules = 10.000.000.000L;
```

```
1 char lettre = 'c';
```

```
1 boolean test = true;
2 test = false;
```

Transtypage ou cast

On a la situation suivante :

```
1 <type1> <nom1> = <valeur1>;
2 <type2> <nom2> = <nom1>;
```

- La conversion ou cast peut rester implicite si le <type1> est « moins fort » que le <type2>

```
1 int i = 10;
2 double x = i;
```

- La conversion doit devenir explicite si le <type1> est « strictement plus fort » que le <type2> : il faut indiquer au compilateur d'effectuer la conversion.

```
1 double x= 3.1416;
2 int i = (int)x;
```

Opérateurs unaires

Opérateur	priorité	action	exemples
+	1	signe positif	`+a; +7`
–	1	signe négatif	`-a; -(a-b); -7`
!	1	négation logique	`!(a<b);`
++		affectation et incrément de 1	`n++;++n;`
——		affectation et incrément de 1	`n++; ——i;`

Opérateurs binaires

Opérateur	priorité	action	exemples
`*`	2	multiplication	`a * i`
`/`	2	division	`n/10`
`%`	3	reste de la division entière	`k%n`
`+`	3	addition	`1+2`
`-`	3	soustraction	`x-5`
`<`	5	strictement inférieur	`i<n`
`<=`	5	inférieur ou égal	`i <= n`
`>`	5	strictement supérieur	`i < n`
`>=`	5	supérieur ou égal	`i >= n`
`==`	6	égalité	`i==j`
`!=`	6	différent	`i!=j`
`&`	7	conjonction (et logique)	`(i<j) & (i<n)`
`|`	9	disjonction (ou logique)	`(i<j) | (i<n)`
`&&`	10	conjonction optimisée	`(i<j) && (i<n)`
`||`	11	disjonction optimisée	`(i<j) || (i<n)`
`=`		affectation	`x = 10; x=n;`
`+=, -=`		affectation et incrément	`i += 2; j-=4`

Exemple

```
1 int i=2, j = i++;
2 i=2;
3 j= ++i;
```

Exemple

```
1 int i=2, j = i++;
2 i=2;
3 j= ++i;
```

- Après l'exécution de la ligne 1 : i=3 et j=2
- Après l'exécution de la ligne 2 : i=2 et j=2
- Après l'exécution de la ligne 3 : i=3 et j=3

Exemple

```
1 int i=2, j = i++;
2 i=2;
3 j= ++i;
```

- Après l'exécution de la ligne 1 : i=3 et j=2
- Après l'exécution de la ligne 2 : i=2 et j=2
- Après l'exécution de la ligne 3 : i=3 et j=3

Attention à ne pas utiliser = pour faire un test d'égalité!

Opérateur conditionnel ternaire

```
1 result = uneCondition ? value1 : value2;
```

Si le test (une expression booléenne) `uneCondition` est vérifié,
alors la variable `result` prend la valeur `value1`,
sinon elle prend la valeur `value2`.

```
1 double x,y, r=1.0;
2    ...
3 boolean interieur = x*x + y*y < r ? true : false
```

Type d'une expression

Le code suivant est-il correct ?

```
1 int i = 5,j;
2 double x = 5.0;
3 j=i/2;
4 j=x/2;
```

Type d'une expression

Le code suivant est-il correct ?

```
1 int i = 5, j;
2 double x = 5.0;
3 j=i/2;
4 j=x/2;
```

- La ligne 3 est correcte et j prendra la valeur 2 (division entière car division entre deux entiers)
- La ligne 4 a une erreur : la division n'est pas entière car `x` est un `double`. Donc le résultat est un `double` et on ne peut pas stocker un `double` dans un `int` sans effectuer de cast !

Type d'une expression

Le code suivant est-il correct ?

```
1 int i = 5, j;
2 double x = 5.0;
3 j=i/2;
4 j=x/2;
```

- La ligne 3 est correcte et j prendra la valeur 2 (division entière car division entre deux entiers)
- La ligne 4 a une erreur : la division n'est pas entière car `x` est un `double`. Donc le résultat est un `double` et on ne peut pas stocker un `double` dans un `int` sans effectuer de cast !

```
1 double x=2.75;
2 int y = (int) x * 2;
3 int z = (int) (x *2);
```

Quelles sont les valeurs de `y` et `z` ?

Les Tableaux

Déclaration

```
1 <type> [] ligne;
2 <type> [][] rectangle;
3 <type> [][][] cube;
```

Création d'un tableau : il **faut** connaître la **taille** du tableau.

```
1 <type> [] ligne = new <type>[<taille1>];
2 <type> [][] rectangle =
3       new <type>[<taille2>][<taille3>];
```

La taille du tableau : `cube.length`

Attention ! le **premier** élément d'un tableau a pour index **0**,
⇨ le **dernier** élément a pour index **length-1**.

Accès aux éléments du tableau avec **[]** :

```
rectangle[3][4] + cube[1][2][5];
```

Exemples

```
1 int[][][] cube = new int[3][4][5];
2 int[][] rectangle = cube[2];
3 int n1 = cube.length;
4 int n2 = cube[0].length;
5 int n3 = cube[0][0].length;
```

Initialisation possible avec une syntaxe de type « liste » :

premiers :

2	3	5	7	11	13	17	19

triangle :

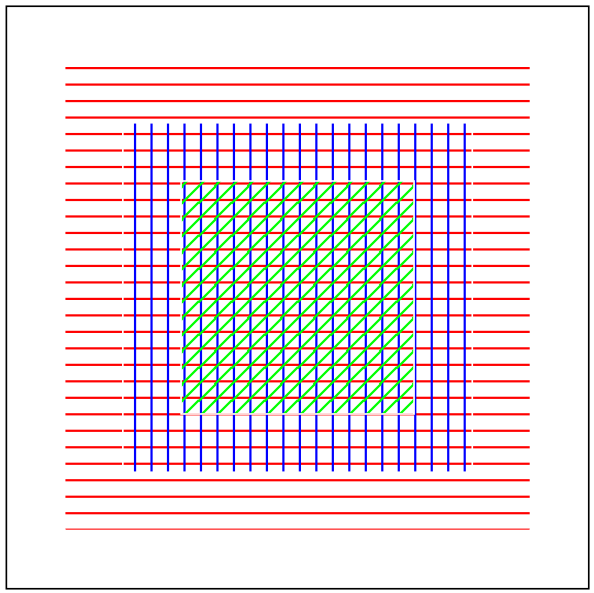
1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1

```
1 int[] premiers = {2, 3, 5, 7, 11, 13, 17, 19 };
2 int[][] triangle = {{1,1,1,1}, { 0,1,1,1},
3       { 0, 0, 1, 1}, {0, 0, 0, 1} };
```

Blocs

Un bloc rassemble des instructions.
Les variables déclarées dans un bloc interne **ne** sont **pas** connues dans un bloc plus externe.

```
1  int a,b=10;
2   {
3     int d=2*b;
4     {
5       int e=b+d;
5       a=e*d;
6       {
5         int g= b+ d*e;
6       }
6     }
7  }
```



`a` et `b` sont connus partout.
`d` est connu seulement dans la partie rouge
`e` est connu seulement dans la partie blue
`g` est connu seulement dans la partie verte

La structure if ... then ... else

```
1  if ( <expression booléenne> )
2     <bloc d'instructions à exécuter
3       si la condition est satisfaite>
4  else
5     <bloc d'instructions à exécuter
6       si la condition n'est pas satisfaite>
```

```
1  int gains,payment,encaissement,invest;
2  // opérations qui modifient la variable gains
3     ...
4  if (gains<0)
5    payment = gains;
6  else if (gains > 10) {
7    encaissement = 10;
8    invest = gains-10;
9  }
10 else
11   encaissement = gains;
```

Choix multiples

```
1  int choix;
2  ...
3  // l'utilisateur modifie la valeur de choix
4  ...
5  switch(choix) {
6    case 1:
7      //instructions pour le choix 1
8      ...
9      break;
10   case 2:
11     //instructions pour le choix 2
12     ...
13     break;
14   default
15     // instructions dans le reste des cas
16     ...
17 }
```

Le `switch` peut s'effectuer sur deux types de variables : `int` et `char`. Depuis la version 7, on peut aussi utiliser une chaîne de caractères

Boucle `for`

```
1 for (<initialisation> ;
2     <condition de poursuite> ;
3     <mise à jour des valeurs>)
4   <bloc d'instructions>
```

que se passe-t-il ?

```
1 for ( ; ; ){
2     // instructions
3 }
```

un exemple classique :

```
0 int n=10;
1 for (int i=0; i< n; i++ ){
2     // instructions
3 }
```

autre exemple

On peut mettre plusieurs initialisations et avoir des conditions de poursuite plus compliquées.

```
0 int n=10;
1 for (int i=0, j=n; j< i; i++; j-- ){
2     // instructions
3 }
```

Boucle `while`

```
1 while(<condition de pousuite>)
2     <bloc d'instructions>
```

Le bloc d'instructions est exécuté **tant que** la `condition` est satisfaite.

un exemple qui va essayer de déterminer si la suite définie par $u : n \to r^n$ converge :

```
1 double epsilon = 0.0000001;
2 double r = 0.75, u=1;
3 while( u-u*r <= -epsilon && u - u* r >= epsilon)
4     u = u * r;
```

Boucle `do ... while`

```
1 do
2     <bloc d'instructions>
3 while(<condition de poursuite>);
```

Attention : ne **pas** oublier le `;` à la fin du `while` !

```
1 double epsilon = 0.0000001;
2 double r = 0.75, u=1;
3 do
4     u = u * r;
5 while ( -epsilon >= u-u*r || u - u* r >= epsilon);
```

choix du type de de la boucle

- généralement, si on connait le nombre d'itérations, on utilise une boucle `for`.
- qu'est-ce qui est plus élégant ?
- qu'est-ce qui sera le plus facile à lire pour un autre lecteur ?

choix du type de de la boucle

- généralement, si on connait le nombre d'itérations, on utilise une boucle `for`.
- qu'est-ce qui est plus élégant ?
- qu'est-ce qui sera le plus facile à lire pour un autre lecteur ?

ex :

- chercher un élément dans un tableau ?
- chercher l'élément le plus grand d'un tableau ?
- déterminer si un nombre est premier ?

choix du type de de la boucle

- généralement, si on connait le nombre d'itérations, on utilise une boucle `for`.
- qu'est-ce qui est plus élégant ?
- qu'est-ce qui sera le plus facile à lire pour un autre lecteur ?

ex :

- chercher un élément dans un tableau ? while
- chercher l'élément le plus grand d'un tableau ?
- déterminer si un nombre est premier ?

choix du type de de la boucle

- généralement, si on connait le nombre d'itérations, on utilise une boucle `for`.
- qu'est-ce qui est plus élégant ?
- qu'est-ce qui sera le plus facile à lire pour un autre lecteur ?

ex :

- chercher un élément dans un tableau ? while
- chercher l'élément le plus grand d'un tableau ? for
- déterminer si un nombre est premier ?

choix du type de de la boucle

- généralement, si on connait le nombre d'itérations, on utilise une boucle `for`.
- qu'est-ce qui est plus élégant ?
- qu'est-ce qui sera le plus facile à lire pour un autre lecteur ?

ex :

- chercher un élément dans un tableau ? while
- chercher l'élément le plus grand d'un tableau ? for
- déterminer si un nombre est premier ? while

Méthodes

But : rassembler une suite d'instructions que l'on répète dans le code.

- en utilisant des méthodes, le code devient
  - plus lisible
  - moins long
- si on veut modifier le code, il n'y a plus qu'**un seul** endroit à changer.

Méthode

```
1 public static <type de retour> <nom>
2       ( <liste de paramètres>) {
3     corps de la méthode : suite d'instructions
4 }
```

Le sens de `public` et `static` seront vus plus tard dans le cours

- l'ordre des arguments est important !
- si la méthode ne retourne rien, son type de retour est **`void`**.
- sinon, on retourne la valeur en utilisant le mot-clé **`return`**.
- choisir un nom de méthodes parlant

Le nom et la liste d'arguments identifient de manière (presque) unique une méthode.
⇨ On appelle **signature** le nom et la liste des arguments d'une méthode.

Exemple

```
1  public static int max( int[] tableau) {
2    int m= tableau[0];
3    for (int i=1;i<tableau.length; i++){
4      if (tableau[i] > m)
5        m = tableau[i];
6    }
7    return m;
8  }
```

Appel de la méthode

```
1  int tab = {7, 12, 15, 9, 11, 17, 13};
2  int m = max(tab);
```

Surcharge

nom de méthode + liste d'arguments = signature

La signature est unique.
⇨ On peut utiliser le même nom mais avoir une liste d'arguments différente
on appelle cette possibilité la **surcharge** de la méthode.

```
1  public static double max( double[] tableau) {
2    double m= tableau[0];
3    for (int i=1;i<tableau.length; i++){
4      if (tableau[i] > m)
5        m = tableau[i];
6    }
7    return m;
8  }
```

on aurait ainsi deux méthodes qui porte le même nom : une méthode qui travaille sur un tableau d'`int` et une autre méthode qui travaille sur un tableau de `double`.

Passage des arguments de types primitifs

```
1 public int f(int n){
2     n = 3 * n * n -2 *n + 1
3    if (n > 0)
4      return n;
5    else
6      return 0;
7 }
```

```
1 int i=13;
2 int j= f(i);
```

Quelle est la valeur de `i` après l'exécution de la ligne 2 ?

Le passage des arguments se fait par valeur (i.e. on copie la valeur de la variable passée en arguments).

Passage des arguments de types primitifs

```
1 public int f(int n){
2     n = 3 * n * n -2 *n + 1
3    if (n > 0)
4      return n;
5    else
6      return 0;
7 }
```

```
1 int i=13;
2 int j= f(i);
```

Quelle est la valeur de `i` après l'exécution de la ligne 2 ?

Le passage des arguments se fait par valeur (i.e. on copie la valeur de la variable passée en arguments).

⇨ après la ligne 2, `i` aura toujours la valeur 13.