Chapitre 1

Structures de programmation de base (1)

Thibaud Martinez

thibaud.martinez@dauphine.psl.eu

Dauphine | PSL
UNIVERSITÉ PARIS

Java "Hello, World!"

```java
public class PremierProgramme {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- Dans un programme Java, tout fait partie d'une classe.
- ⚠️ Le nom du fichier doit être le même que celui de la classe publique, avec l'extension `.java`, soit `PremierProgramme.java`.
- `main` est le point d'entrée du programme. L'exécution du code débutera par cette méthode.

Commentaires

```
// ceci est un commentaire sur une seule ligne
```

```
/* ceci est un commentaire
sur plusieurs
lignes */
```

Expressions et instructions

- Une **expression** est une combinaison de variables, d'opérateurs et d'appels de méthodes construite selon la syntaxe du langage et qui s'évalue à une seule valeur. Par exemple `1 * 2 * 3`.
- Une **instruction** constitue une unité d'exécution complète.
  Par exemple `int i = 10;`

  Les instructions se terminent par un `;`.

Types de données

Java est un langage **fortement typé** chaque variable doit avoir un type déclaré.

Il existe **8 types élémentaires**.

Type	Taille (bits)	Description	Valeurs
`boolean`	1	Une valeur logique	`true` ou `false`
`byte`	8	Un octet signé	-128 à 127
`char`	16	Un caractère Unicode	\u0000 à \uFFFF
`short`	16	Un entier court signé	-32768 à 32767
`int`	32	Un entier signé	-2^{31} à $2^{31}-1$
`long`	64	Un entier long	-2^{63} à $2^{63}-1$
`float`	32	Un nombre à virgule flottante de 32 bits signé (simple précision)	2^{-149} à $(2-2^{-23)}\cdot 2^{127}$
`double`	64	Un nombre à virgule flottante de 64 bits signé (double précision)	2^{-1075} à $(2-2^{-52})\cdot 2^{1023}$

Variables

Déclaration

```
<type> <nom>;
```

```
double salaire;
int jourDeVacances;
long population;
boolean doitContinuer;
```

✨ Il est d'usage d'écrire le nom des variables en ***lower camelCase***.

Initialisation

Après avoir **déclaré** une variable, on doit l'**initialiser** au moyen d'une instruction d'affectation.

```
int jourDeVacances;   // déclaration
jourDeVacances = 12;  // initialisation
```

```
int jourDeVacances = 12;
```

⚠️ On ne peut pas utiliser la valeur d'une variable non initialisée.

Inférence de type

Depuis Java 10, le compilateur peut déduire le type de la variable de sa valeur d'initialisation.

```
var taux = 4.29;    // taux est un double
var lettre = 'A';   // lettre est un char
```

JEP 286: Local-Variable Type Inference

Constantes

Les constantes sont des variables dont la valeur ne change pas.

On utilise le mot clé `final` pour définir une constante.

```
final <type> <nom>;
```

```
final double PI = 3.141592;
```

✨ On écrit le nom des constantes en SCREAMING_SNAKE_CASE.

Transtypage ou *cast*

Il s'agit de convertir le type d'une variable ou une valeur donnée.

- la conversion peut avoir lieu **implicitement** si le type de originel est "moins fort" que le type visé.

```
int x = 9;
double y = x;
```

- dans les autres cas, la conversion doit être **explicite**.

```
double a = 6.56;
int b = (int)a;
```

Opérateurs

Les opérateurs permettent de combiner des valeurs.

Opérateurs arithmétiques

- addition : `2 + 3`
- soustraction : `2 - 3`
- multiplication : `2 * 3`
- division : `2 / 3`
- modulo (reste de la division entière): `5 % 2`

⚠️ L'opérateur `/` désigne une division entière si les deux arguments sont des entiers, et une division en virgule flottante sinon.

Il est courant de réaliser une opération avec la valeur d'une variable puis d'affecter le résultat de cette opération à cette même variable.

Les opérateurs `+=`, `-=`, `*=`, `/=`, `%=` répondent à ces cas d'usage.

```
int i = 1;

i += 2  // i vaut 3
i -= 1  // i vaut 2
i *= 5  // i vaut 10
i /= 2  // i vaut 5
i %= 2  // i vaut 1
```

Incrémentation et décrementation

Ajouter ou retirer 1 à une variable est une opération courante en programmation, il existe les opérateurs unaires `++` et `--` pour réaliser ces opérations.

```
int n = 12;
n++; // n vaut 13
n--; // n vaut 12
```

Opérateurs *prefix* et *postfix*

`++` et `--` peuvent être placés avant la variable (opérateur *prefix*) ou après (opérateur *postfix*). Dans le premier cas la valeur de la variable est modifiée avant d'être renvoyée, dans le second, elle est modifiée après.

```
int m = 7;
int n = 7;
int a = 2 * ++m; // a vaut 16 et m vaut 8
int b = 2 * n++; // b vaut 14 et n vaut 8
```

Opérateurs booléens

- égalité : `x == 9`
- inégalité : `x != 9`
- inférieur : `x < 9`
- inférieur ou égal : `x <= 9`
- supérieur : `x > 9`
- supérieur ou égal : `x >= 9`
- et : `y && z`
- ou : `y || z`
- négation logique : `!x`

Évaluation *short-circuit*

Les opérateurs `&&` et `||` sont évalués selon une logique *"short-circuit"*: le deuxième argument n'est évalué que si le premier argument ne suffit pas à déterminer la valeur de l'expression.

Ce comportement peut être utilisé pour éviter les erreurs.

```
x != 0 && 1 / x > x + y     // pas d'erreur de division par 0
```

Opérateur ternaire

```
condition ? expression1 : expression2
```

L'expression prend la valeur de l'`expression1` si la `condition` est vraie et celle de l'`expression2` sinon.

Exemple

```
int valeur = (1 < 2) ? 3 : 4;   // valeur = 3
int minimum = x < y ? x : y;    // minimum entre x et y
```

Expressions *switch*

Pour choisir entre plusieurs valeurs, on peut utiliser une expression *switch* (à partir de Java 14).

```
int codeTaille = 2;

char taille = switch (codeTaille) {
    case 0 -> 'S';
    case 1 -> 'M';
    case 2 -> 'L';
    default -> '?';
};
// taille = 'L'
```

Opérateurs *bitwise*

Ces opérateurs agissent directement sur les **bits**.

- complément : `~` → fait de chaque `0` un `1` et de chaque `1` un `0`.
- décalage à gauche signé : `<<` → décale la séquence de bits vers la gauche
- décalage à droite signé : `>>` → décale la séquence de bits vers la droite
- et : `&`
- ou exclusif (XOR) : `^`
- ou inclusif (OR) : `|`

Ils sont relativement peu utilisés et servent essentiellement pour optimiser les calculs.

Structures de contrôle

Bloc et portée des variables

Un **bloc** consiste en un certain nombre d'instructions Java, entourées d'une paire d'**accolades**.

Les blocs définissent la **portée** des variables.

```
public static void main(String[] args) {
    int n;
    {
        int k;
        // ...
    } // k est défini et utilisable jusqu'à ce point
}
```

⚠️ Vous ne pouvez pas déclarer des variables locales de même nom dans deux blocs imbriqués.

```
public static void main(String[] args) {
    int n;
    {
        int k;
        int n; // Error: variable n is already defined...
    }
}
```

Instructions conditionnelles

Elles permettent d'exécuter un bloc d'instructions si une condition est remplie.

```
if (<condition>) {
    <instruction>
}
```

```
if (ventes >= objectif) {
    bonus = 100;
}
```

if-else

Si la condition est remplie ... **sinon** ...

```
if (<condition>) {
    <instruction>
} else {
    <instruction>
}
```

```
if (ventes >= objectif) {
    bonus = 100;
else {
    bonus = 0;
}
```

if-else-if

Si la condition est remplie ... **sinon si une autre condition** est remplie ...

```
if (vente >= 2 * objectif) {
    bonus = 1000;
} else if (vente >= objectif) {
    bonus = 100;
}
```

switch

La construction *if-else* peut s'avérer lourde lorsqu'on doit traiter plusieurs alternatives pour la même expression. *switch* offre une alternative plus compacte.

```
int codeTaille = 2;

switch (codeTaille) {
    case 0 -> { System.out.println('S'); }
    case 1 -> { System.out.println('M'); }
    case 2 -> { System.out.println('L'); }
    default -> { System.out.println('?'); }
};
```

✨ Lorsque c'est possible, on préfera utiliser des expressions *switch* (voir diapositive précédente).

Boucle *while*

La boucle *while* exécute une instruction **tant qu'une condition est vraie**.

```
while (<condition>) {
    <instruction>
}
```

⚠️ La boucle while ne s'exécutera jamais si la condition est fausse au départ.

```
while (solde < objectif) {
    solde += paiement;
    double interets = solde * tauxInteret;
    balance += interets;
    annees++;
}
```

Boucle *do-while*

Pour s'assurer qu'un bloc d'instructions est exécuté au moins une fois.

```
do {
    <instruction>
} while (<condition>);
```

```
do {
    solde += paiement;
    double interets = solde * tauxInteret;
    solde += interet;
    years++;
}
while (solde < objectif);
```

Boucle *for*

```
for (<initialisation>; <condition d'arrêt>; <mise à jour>) {
    <instruction>
}
```

```
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}
```

Break

L'instruction *break* est utilisée pour sortir d'une boucle.

```
while (annees <= 100) {
    solde += paiement;
    double interets = solde * tauxInteret;
    solde += interet;

    if (solde >= objectif) {
        break;
    }

    years++;
}
```

Méthodes

Une méthode Java joue un rôle similaire à une fonction dans un langage comme C. Il s'agit d'une séquence d'instructions qui effectue une tâche spécifique, empaqueté comme une unité et qui renvoie une valeur.

```
public static <type de retour> <nom> (<paramètres>) {
    <instructions>
}
```

Le sens des mots clés `public` et `static` sera vu plus tard dans le cours.

Si la méthode ne renvoie rien, son type de retour est `void`.

```
public static void afficheNombre() {
    System.out.println(42)
}
```

Pour retourner une valeur, on utilise le mot-clé `return`.

```
public static int additionne(int a, int b) {
    return a + b;
}
```

L'appel de la méthode s'effectue de la façon suivante :

```
int resultat = additionne(9, 14)
```

`9` et `14` constituent les **arguments** passés lors de l'appel de la méthode.

Signature de la méthode et surcharge

Une méthode est identifiée par son **nom** et le **nombre, type et ordre de ses paramètres**. Ces éléments définissent la **signature de la méthode**.

Ainsi, deux fonctions peuvent avoir le même nom mais des paramètres différents.

```
public static int additionne(int a, int b) {
    return a + b;
}

public static double additionne(double a, double b) {
    return a + b;
}
```

C'est ce qu'on appelle la **surcharge des méthodes**.

Passage par valeur

Le passage des arguments de type élémentaire se fait **par valeur**, c'est-à-dire que la méthode n'opére pas sur les valeurs des variables en arguments mais sur des **copies des valeurs**.

```
public static void mtd(int n) {
    n = n + 1;
}

int v = 10;
mtd(v)
// v vaut toujours 10
```