Introduction

Amélie Lambert

USSE09 - Java

Planning du cours

Date	heure	type	EMBD	Programme
12/03/24	8 :15	CM/ED	C. Rambour	Structure contrôles et types
12/03/23	13:30	TP	C. Rambour	Structure contrôles et types
19/03/24	8 :15	CM/ED	A. Lambert	Sous-programmes et tableaux
19/03/24	13:30	TP	A. Lambert	Sous-programmes et tableaux
26/03/24	08:15	CM/ED	A. Lambert	Programmation par Objet (1/2)
26/03/24	13:30	TP	A. Lambert	Programmation par Objet (1/2)
02/04/24	08:15	CM/ED	A. Lambert	Programmation par Objet (2/2)
02/04/24	13:30	TP	A. Lambert	Programmation par Objet (2/2)
23/04/24	13:30	CM/ED	C. Rambour	Programmation Orientée Objet (1/2)
23/04/24	13:30	TP	A. Lambert	Programmation Orientée Objet (1/2)
30/04/24	8:15	CM/ED	C. Rambour	Programmation Orientée Objet (2/2)
30/04/24	13:30	TP	A. Lambert	Programmation Orientée Objet (1/2)
13/05/27	8:15	CM/ED	A. Lambert	Exceptions
14/05/24	8:30	TP	A. Lambert	Exceptions
14/05/24	13:30	CM/ED	A. Lambert	Projet
28/05/24	8:15	TP	A. Lambert	Projet
29/05/24	8:15	TP	A. Lambert	Projet
29/05/24	13:30	TP	A. Lambert	Projet
31/05/24	**10:15**	**DS**	**A. Lambert**	

Supports de cours, TD et TP sur https://lecnam.net

Modalités d'évaluation

- Tous les TPs sont à rendre

TP	TP	Date de rendu
TP 1	TP1 : Structures de contrôles et types	19/03/24
TP	TP2 : Sous-programmes et tableaux	26/03/24
TP	TP3 : Programmation par objet	23/04/24
TP	TP4 : Programmation orientéé objet	14/05/24
TP	TP5 : Exceptions	21/05/24
TP	PROJET	02/06/24

- Un DS
- Un projet

Note finale : $\frac{0.25*TP+0.75*DS+Projet}{2}$

Choix du Java

Points forts du Java

- C'est un langage simple qui hérite des constructions de C et C++
- C'est un langage **orienté objet** qui permet la conception et la réalisation d'applications complexes avec une architecture modulaire
- C'est un langage **robuste** muni d'un mécanisme de gestion des exceptions qui permet de déceler et de traiter des erreurs pendant l'exécution du programme
- Java est **indépendant de la plate forme** d'exécution, donc indépendant de la machine et de son système d'exploitation
- Java est un langage **distribué**. Un programme peut être déployé sur plusieurs machines d'un réseau d'ordinateurs et exécuté sur celui-ci

Java Application Programming Interface (API)

- Application Program Interface (API) est un ensemble de classes et interfaces prédéfinies

- 3 éditions d'API
  - **J2SE** pour le développement d'applications coté client
  - **J2EE** pour le développement d'applications coté serveur
  - **J2ME** développements pour mobiles

Outils de développement

Ces outils fournissent un environnement de développement intégré dans une interface graphique (Integrated Development Environment ou IDE).

Principaux IDE :

- JBuilder (Borland)
- NetBeans Open Source (Sun)
- Sun One (version commerciale de NetBeans)
- Eclipse Open Source (IBM)

Environnement Java

J2SE (Java 2 Standard Edition) actuellement J2SE 17

- comprend un JDK (Java Development Toolkit), ensemble de programmes invocables à partir d'une ligne de commande, dont :
  - `javac` : compilateur
  - `java` : JVM interpréteur de `bytecode`
  - `javadoc` : générateur de documentation.
- Rassemble :
  - Environnement d'exécution
  - Langage
  - Application Programming Interfaces
  - Bibliothèques (`.jar`)

La chaine de production de programme

Les programmes

Qu'est ce qu'un programme ?

- C'est la description d'une méthode de resolution d'un problème donné.
- Cette description est donnée par une suite d'instructions qui traitent les données du problème à resoudre, jusqu'à aboutir à une solution.

Exécution d'un programme

- Les traitements décrits par les instructions sont appliqués aux données (entrées),
- Les données ainsi transformées permettent d'aboutir aux solutions recherchées (sorties).

Les langages de programmation

Les langages de haut niveau : (Ex : C, Ada, Pascal, Cobol, Java, OCaml, Python).

- fournissent des constructions sophistiquées qui facilitent l'écriture des programmes.
- sont comprehensibles par les humains, mais pas directement exécutables par les machines.
- $\Rightarrow$ Traduction en langage machine avant son exécution.

Les langages de bas niveau : (langages cibles ou natifs)

- instructions propres à chaque machine (SPARC/Sun, Intel/PC, etc).
- codés en binaire et directement exécutables par chaque machine.

Description d'un langage de haut niveau

Les types des données : utilisés pour modéliser les données.
Exemple : Le type `int` en Java sert à modéliser les nombres entiers.

La syntaxe : règles de formation textuelle des instructions.
Exemple : pour écrire l'expression mathématique $1 \leq x \leq 7$, une syntaxe possible en Java est : `1 <= x && x <= 7`.

La sémantique : règles qui précisent "le sens" des contructions syntaxiques à l'exécution et la cohérence entre types.
Exemples :

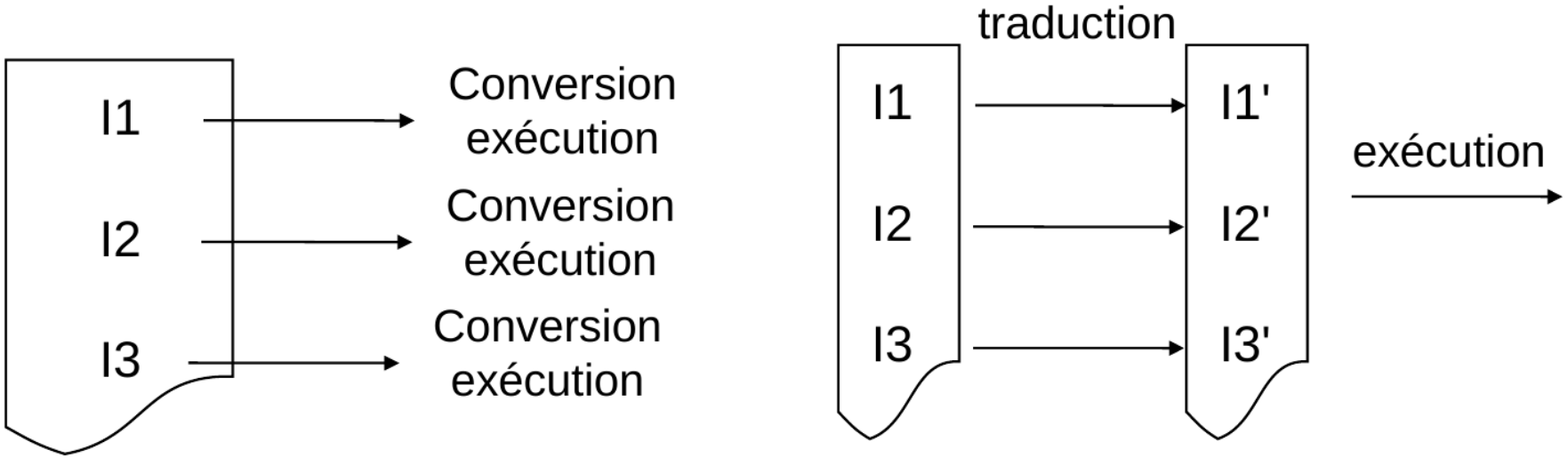
- `4+3*2` équivaut à la valeur entière 10
- `"bonjour"*2` est correct syntaxiquement, mais non sémantiquement : `*` ne peut s'appliquer à une chaîne de caractères

Compilation vs Interprétation

Rôle : traduction du langage de haut niveau vers du langage machine spécifique à un processeur ⇒ il peut lire les instructions et les exécuter.

2 méthodes :

- `Compilation :` traduction de toutes les instructions, puis exécution de la traduction : le résultat dépend de l'ordinateur visé
- `Interprétation :` conversion et exécution de chaque instruction les unes derrière les autres : plus fexible mais plus lent.



En java : Compilation suivie d'interprétation

Idée : placer un intermédiaire dans la phase de compilation

1. Compiler vers un langage machine de la machine virtuelle Java (JVM) : le `bytecode`.
   $\Rightarrow$ Utilisation de la commande `javac`
2. Un programme, appelé interpréteur, traduit au vol (pendant l'exécution même du programme) le `bytecode` en instructions pour le processeur.
   $\Rightarrow$ Utilisation de la commande `java`

Avantage : Le `bytecode` est indépendant de tout ordinateur

Structure d'un programme Java

Structure d'un programme Java

En java, chaque fichier est une **classe** (mots clé `class`).
Un programme a au moins une classe contanant une méthode `main` qui recoit la suite d'instructions à exécuter.

```
public class Hello{
  public static void main( String[] args ){
     System.out.println("Bonne "+args[0]+" "+args[1]+" "+"2022");
  }
}
```

Remarques

- Le nom du programme est le même que celui de la classe principale (ici `Hello`)
- Le nom du fichier qui contient une classe est son nom suivit de l'extension `.java` (ici `Hello.java`)
- Par convention, le nom d'une classe commence par une majuscule.

Structure d'un programme Java

```
public class Hello{
  public static void main( String[] args ){
     System.out.println("Bonne "+args[0]+" "+args[1]+" "+"2022");
  }
}
```

`main` reçoit les arguments de la commande de lancement `String[] args`

Production de programme

1. `Compilation :`
   `javac Hello.java`
2. `Exécution :`
   `java Hello nouvelle année`
   avec `args[0] = nouvelle` et `args[1] = année`
3. `Résultat affiché :`
   `Bonne nouvelle année 2022`

Les types de données

Notion de type de données

- Toutes les variables ont un **type**.
  Le type de la variable définit la taille de l'espace mémoire réservé pour celle-ci.

- Un type définit l'ensemble des valeurs que peut prendre un objet informatique ainsi que les opérations permises sur ces valeurs

  Définition d'un objet informatique :
  Toute entité à laquelle un type est associé (variable, constante, fonction, procédure, exception, paquetage, ...) est un objet informatique.

- **Exemple : le type entier `int` :**
  - Ensemble de valeurs : $\mathbb{N}$
  - Ensemble d'opérations sur ces valeurs $(+, -, *, /, ==, \ldots)$

Intérêt du concept de type

Fournir à la machine les **informations** nécessaires pour qu'elle soit en mesure de **vérifier que l'utilisation** des objets d'un programme est bien **conforme** à l'intention préalable du programmeur :

- Lorsque nous déclarons un **objet (variable, constante, fonction, ...)**, nous spécifions les contraintes de son utilisation, l'ensemble des valeurs qui pourront lui être associées et ainsi le **cadre légitime de son utilisation**.
- L'utilisation de cet objet sera précédée d'une **vérification effectuée par le compilateur**. La valeur qu'il prend appartient-elle à cet ensemble ? Son utilisation est-elle conforme aux contraintes spécifiées ?

Le concept de type permet d'établir un lien sémantique entre la déclaration et l'utilisation d'un objet.

Les types primitifs

Type	Ensemble des valeurs	Taille
`byte`	$[-128, \ldots, 127]([-2^7, \ldots, 2^7 - 1])$	1 octet
`short`	$[-32768, \ldots, 32767]([-2^{15}, \ldots, 2^{15} - 1])$	2 octets
`int`	$[-2^{31}, \ldots, 2^{31} - 1]$	4 octets
`long`	$[-2^{63}, \ldots, 2^{63} - 1]$	8 octets
`float`	réels simple précision (7 chiffres significatifs)	4 octets
`double`	réels double précision (15 chiffres significatifs)	8 octets
`boolean`	`true`, `false`	1 octets
`char`	les caractères	2 octets

Valeurs littérales numériques

valeur	2	2.0	2.0f ou 2.0F	2.0d ou 2.0D	2l ou 2L	2f ou 2F	2d ou 2D
type	`int`	`double`	`float`	`double`	`long`	`float`	`double`

Un entier suivi de `f (F)` devient un `float`

Un entier suivi de `d (D)` devient un `double`

Opérateurs arithmétiques

- **Opérateurs sur les types entiers** (`byte`, `short`, `int`, `long`)
  - `+`, `-`, `*`, `/`, `%`
  - Exemples : `9/4 = 2`, `9%4 = 1`
- **Opérateurs sur les types réels** (`double`, `float`)
  - `+`,`-`,`*`,`/`
  - Exemple : `9/4 = 2,25`
- **Opérateurs relationnels**
  - `<`, `>`, `<=`, `>=`, `==`, `!=`
  - Exemples : `5 > x`, `a == 'b'`, `(a != 0) && (a <= 100)`

Précédence des opérateurs (1/2)

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` $\Rightarrow$ équivaut à `2 +(3*4)`
  $\Rightarrow$ s'évalue en 14 et non pas en 20
- `2 + 3 > 4` $\Rightarrow$ s'évalue en `true`
- `2 + (3 > 7)` $\Rightarrow$ produit une erreur

Précédence des opérateurs (2/2)

De la plus haute à la plus basse priorité :

```
+, -
  !
* / %
 + -
< <= > >=
 == !=
  &&
  ||
```

L'opérateur conditionnel ternaire ? :

`z = (x>y)? x : y;` $\iff$ `if (x>y) z = x; else z = y;`

- 3 opérandes : `(x>y)`, `x` et `y`
- Si la première opérande vaut true, l'expression a pour valeur celle de x sinon elle a pour valeur celle de y

Exemple :

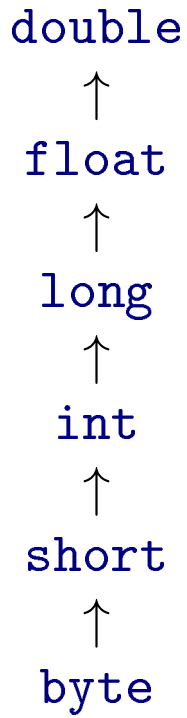
```
int x = ......;
System.out.println(
    (x%2==0)? x+" est pair" : x+" est impair")
```

Opérateurs et expressions

- Un **opérateur** permet de réaliser des calculs sur une ou plusieurs valeurs appelées **opérandes**.
- Il existe des opérateurs unaires (une seule opérande), binaires et ternaires (ex : `!true`, $9 * X$)
- Les **expressions** sont des constructions décrivant des opérations à l'aide d'opérateurs et d'opérandes
- Les opérandes d'une expression peuvent être des constantes, des variables valuées ou des appels de fonction :
  - `(34 + Math.min(X,Y))` où `X` et `Y` sont les arguments valués de la fonction `min` de la classe `Math`
  - $2 > 5$ est une expression booléenne dont le résultat est `false`
  - $!(2 > 5)$ est une expression booléenne lue : `non 2 plus grand que 5`
- Une expression correspond toujours à la valeur qu'elle calcule

Conversions implicites entre types numériques

- Les opérandes sont de même type $\Rightarrow$ le résultat est du même type
- Pour que l'opération soit réalisée, il faut que toutes les opérandes soient de même type
  $\Rightarrow$ nécessité de convertir certaines opérandes. La conversion se fait vers le type le plus large selon la hiérarchie :



Exemple

- `6.2f` $-$ `4` $\Rightarrow$ `6.2f` $-$ `4.0f` : le résultat est de type `float`.
- `5.4f` $+$ `55.8` $\Rightarrow$ `5.4` $+$ `55.8` : le résultat est de type `double`.
- **Remarque :** aucune perte d'information

Conversions explicites entre types numériques

- Une conversion explicite est appelée **"cast"** en Java
- **syntaxe :** `(type_cible)val`
  - `type_cible` est le type vers lequel on fait la conversion
  - `val` la valeur à convertir
- **Exemples :**
  - `(int)`64.89 $\Rightarrow$ 64 on passe d'une représentation sur 8 octets vers une à 4 octets avec changement de codage de la valeur
  - `(float)`64.89 $\Rightarrow$ on passe d'une représentation sur 8 octets vers une à 4 octets sans changement de codage de la valeur

Le type Boolean

- Ensemble des **valeurs** du type : `true`, `false`
- Ensemble des **opérateurs** : `&&`, `||`, `!` :
  - && conjonction, se dit "et"
  - || disjonction, se dit "ou"
  - ! négation, se dit "non"
- **Exemples :**
  - `true && false` $\Rightarrow$ `false`
  - `!x || (b && c)`
  - $('a' <' y' \;\&\&\; 'b' >' x') \Rightarrow$ `false`
  - $(X\%400 == 0) \;||\; (X\%4 == 0 \;\&\&\; X\%100 != 0) \Rightarrow$ `true` si `X` est une année bissextile.

Tables de vérité

p	q	p && q
v	v	v
v	f	f
f	v	f
f	f	f

Conjonction

p	q	p || q
v	v	v
v	f	v
f	v	v
f	f	f

Disjonction

p	q	p ^ q
v	v	f
v	f	v
f	v	v
f	f	f

Disjonction exclusive

Le type `String`

Le type `String`

- Le type `String` n'est pas un type primitif. Ses valeurs sont des objets représentant des chaînes de caractères.

- Les valeurs du type `String` s'écrivent entre quotes :
  - `"hello" "123" "__XX_"`
  - `"mon nom est : \"Martin\"!"`

- **Opérateur de concaténation : +**

  La chaîne : `"hello"+" world"` est identique à `"hello world"`

Utilisation du type String

- **Syntaxe :**

  `<chaine>.<ident_méthode>(<liste_param>)`

  `<chaine>` est une valeur du type `String`

  `<ident_méthode>` est un nom de méthode applicable sur les valeurs du type String

  `<liste_param>` le ou les paramètres séparés par une virgule

- **Exemple :**

```
int i = "toto".length; // i ← 4
```

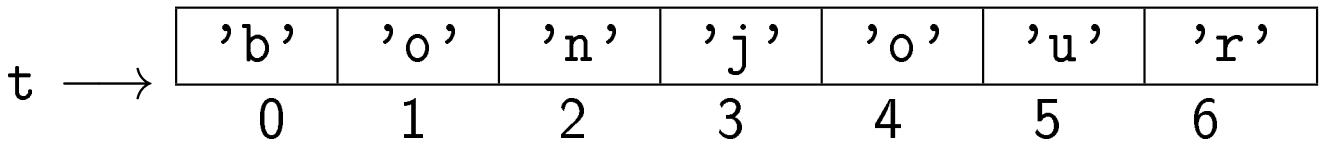
Quelques méthodes du type String

- `char charAt( int n)` retourne le `n`-ième caractère de la chaîne

  Exemple : `char c = "bonjour".charAt(0); // c` $\leftarrow$ `'b'`

- `char[] toCharArray()` transforme une chaîne en un tableau de caractères

  Exemple : `char[] t = "bonjour".toCharArray();`



- `String toLowerCase()`, `String toUpperCase()` pour transformer les caractères en majuscules ou en minuscules

Comparaison de chaînes

`int compareTo(String s2)`

Compare une chaîne `s1` avec une chaîne `s2` selon l'ordre lexicographique.

- `s1 < s2` retourne un entier `< 0`
- `s1 = s2` retourne un entier `= 0`
- `s 1> s2` retourne un entier `> 0`

Exemple :

```
String s = "SOS", t = "HELP";
if (s.compareTo(t) > 0)
  System.out.println( t+" > "+s);
else
  System.out.println( t+" <= "+s);
```

MAIS attention :

```
if (s > t)     // ILLEGAL
```

Egalité entre chaînes (1/2)

`boolean equals(String s2)`

Teste l'égalité d'une chaîne `s1` avec une chaîne `s2`, renvoie `true` ou `false`

Exemple :

```
String s1 = "SOS";
String s2 = "OSS";
if( s1.equals(s2))
  System.out.println( s1+" = "+s2);
else
  System.out.println( s1+" != "+s2);
```

Affichage : `SOS != OSS`

```
String s1 = "SOS";
String s2 = "SOS";
if( s1.equals(s2) )
  System.out.println( s1+" = "+s2);
else
  System.out.println( s1+" != "+s2);
```

Affichage : `SOS = SOS`

Egalité entre chaînes (2/2)

On distingue 2 sortes d'égalité : l'**égalité de contenant (l'adresse d'objets)** et l'**égalité de contenu**.

Les opérateurs : == et != testent si deux noms de variables différents désignent le même **contenant**.

Exemple :

```
String s1 = "SOS";
String s2 = s1;
if( s1==s2)
    System.out.println( s1+" = "+s2);
else
    System.out.println( s1+" != "+s2);
```

Affichage : s1 = s2

```
String s1 = "SOS";
String s2 = "SOS";
if( s1==s2)
    System.out.println( s1+" = "+s2);
else
    System.out.println( s1+" != "+s2);
```

Affichage : s1 != s2

Conversion vers les types primitifs

- `String` → `int`
  Exemple :
  ```
  String s = "123";
  int x = Integer.parseInt(s); // x ← 123
  ```
- `String` → `int`
  Exemple :
  ```
  String s = "123";
  double x = Double.parseDouble(s); // x ← 123
  ```
- `String` → `boolean`
  Exemple :
  ```
  String s = "true";
  boolean x = Boolean.parseBoolean(s); // x ← true
  String s = "toto";
  boolean x = Boolean.parseBoolean(s); // x ← false
  ```
- Idem pour `String` → `float`, `String` → `short` ...

Conversion vers le type String

- `int` → `String`
  Exemple :
  ```
  int i = 15;
  String s = ""+15; // s ← "15"
  ```

- `double` → `String`
  Exemple :
  ```
  double i = 15.23;
  String s = ""+15.23; // s ← "15.23"
  ```

- `boolean` → `String`
  Exemple :
  ```
  boolean i = true;
  String s = ""+true; // s ← "true"
  ```

Les structures de contrôle

Structures de contrôle

- La seule instruction qui a des effets sur le contenu de la mémoire est **l'affectation**.
- Les autres instructions ne sont là que pour permettre d'organiser de manière à la fois plus concise et plus lisible la suite des affectations.
- On appelle ces instructions des **structures de contrôle**.
- On distingue les structures de contrôle **conditionnelles** et **itératives**.

Conditionnelle

Syntaxe :

```
<conditionnelle>::=
     if(<expression_booléenne>)
          <suite_instructions>
     [ else <suite_instructions> ]
```

Sémantique :

- Les parties entre crochets sont optionnelles
- Si la condition est vraie, la suite d'instructions associée est exécutée et l'instruction conditionnelle est terminée
- Si la condition est fausse, la partie `else` est exécutée et l'instruction conditionnelle est terminée

Expression conditionnelle ternaire

Syntaxe :

```
<expression_conditionnelle>::=
        <expression_booléeenne> ? <expression> : <expression>
```

Exemples :

```
int y = (x>0) ? 1 : -1
// équivalent à
int y;
if (x>0) y=1; else y=-1;

System.out.println((num%2==0) ? "num pair":"num impair");
// ? est le seul opérateur ternaire en Java
```

Choix multiple

Syntaxe :

```
<choix>::=
switch(<expression>){
        case <valeur>:<instructions>;break;
        case <valeur>:<instructions>;break;
        case <valeur>:<instructions>;break;
        [default : <instructions>;]
}
```

<expression> est obligatoirement de type char,byte,short ou int
<valeur> est une constante de même type que <expression>
<instructions> est une suite d'instructions simples

Structure itérative : boucle `while`

Syntaxe :

```
<boucle>::=
      while(<condition>){
            <suite_instructions>;
      }
```

Sémantique :

`<condition>` est une expression booléenne évaluée avant chaque itération

- si `<condition>` est vrai, `<suite_instructions>` est exécutée, puis de nouveau la condition est évaluée
- si `<condition>` est faux, le contrôle passe à l'instruction qui suit la boucle

Structure itérative : boucle do...while

Syntaxe :

```
<boucle>::=
      do{
            <suite_instructions>
      }while(<condition>);
```

Sémantique :

1. <suite_instructions> est exécutée
2. <condition> est une expression booléenne évaluée à la fin de chaque itération
   - si <condition> est vrai, le contrôle revient à <suite_instructions>
   - si <condition> est faux, le contrôle passe à l'instruction qui suit la boucle

Boucle for

Syntaxe :

```
<boucle_for>::=
for( <init_inst >;<expr_bool>;<modif_inst>){
      <suite_instructions>
}
```

<init_inst> sont les instructions d'initialisation de la boucle séparées par des virgules

<expr_bool> est une expression booléenne, condition de continuation de l'itération

<modif_inst> sont les instructions de mise à jour des variables de la boucle séparées par des virgules

Exemple : boucle infinie

```
for(;;){
   <suite_instructions>
}
```

équivalent à

```
while(true){
   <suite_instructions>
}
```

Rupture de contrôle : break (1/2)

```
public class TestBreak{
  public static void main(String[] args){
  int n = 1;

  for(;;){
    System.out.println("A");

    if ( n>=1 ){
      System.out.println("B");
      break;
    }
    else
      System.out.println("C");
    System.out.println("D");

  }
  System.out.println("F");
 }
}
```

Rupture de contrôle : break (2/2)

Résultat affiché :

```
A
B
F
```

L'instruction `break` permet une sortie immédiate de la boucle la plus interne et non d'un bloc quelconque.

Instruction continue

```
public class TestContinue{
  public static void main(String[] args){
    int somme = 0;
    int n = 0;
    while( n<30 ){
        n++;
        if( n==10||n==12 ) continue;
        somme = somme+n;
    }
    System.out.println("n="+n);    // n=30
    System.out.println("somme="+somme);  // somme =443
  }
}
```

L'instruction `continue` termine le pas d'itération en cours pour commencer le suivant.

Dans l'exemple, si n=10 ou 12, l'instruction `somme = somme+n;` n'est pas exécutée. L'itération n'en est pas pour autant terminée.

Appel de méthode

Syntaxe

```
<ident_sous_programme>(<arg>,<arg>,...,<arg>)
```

- Le sous-programme est invoqué en lui passant les données (arguments) à traiter.
- Les arguments peuvent changer à chaque appel
- Comme résultat, on obtient une valeur ou bien une modification de l'état de la machine appelée effet de bord ou bien encore les deux.

Transmission de paramètre

- Lors de l'appel d'une fonction, le paramètre effectif est transmis au paramètre formel.
- Tout se passe comme si une **copie de la valeur** du paramètre effectif initialisait le paramètre formel
- On parle de **passage par valeur**
- **Conséquence :** l'exécution de la fonction ne peut pas modifier le paramètre effectif
- Le paramètre formel `x` est une **variable** pendant toute l'exécution de la procédure
- **Note :** la valeur transmise est une référence si le paramètre est un tableau ou un objet

Notion de bloc

- Un bloc est une suite d'instructions contenue entre accolades
  Exemples : Le corps d'une boucle (`for`, `while`) est donc un bloc.
  Les instructions entre les accolades d'un `if`, d'un `else` sont des blocs.

- Un bloc est un **environnement local** de déclarations de variables. Les variables n'existent que dans le bloc dans lequel elles sont déclarées

Exemples :

```
public static void main(String[] args){
    int a=5;
    if (a==0){
        int b=3+2*a;
        System.out.println("b="+b);
    }
    else{
        int c=3+a;
        System.out.println("c="+c); // b est inconnu
    }
    System.out.println("a="+a); // b et c sont inconnus
}
```