ITI 1521. Introduction à l'informatique II

Types de données : opérateurs et appels de méthodes

by

Marcel Turcotte

Version du 19 janvier 2020

Préambule

Préambule

Aperçu

Aperçu

Types de données : opérateurs et appels de méthodes

Nous examinons les avantages des langages fortement typés. Nous comparons les types primitifs et les types références au niveau des opérateurs de comparaison et des appels de méthodes.

Objectif général :

- Cette semaine, vous serez en mesure de contraster les types primitifs et les types référence au niveau des opérateurs de comparaison et des appels de méthodes.

Une vidéo d'introduction :

- `https://www.youtube.com/watch?v=JkJwgschr7c`

Préambule

Objectifs d'apprentissage

Objectifs d'apprentissage

- **Comparer** l'évaluation des expressions de type primitif et référence.
- **Énumérer** les étapes d'un appel de méthode.
- **Comparer** les appels de méthodes selon le cas où les paramètres sont de type primitif et le cas où les paramètres sont de type référence.

Lectures :

- Pages 545-551, 571-572 de E. Koffman et P. Wolfgang.

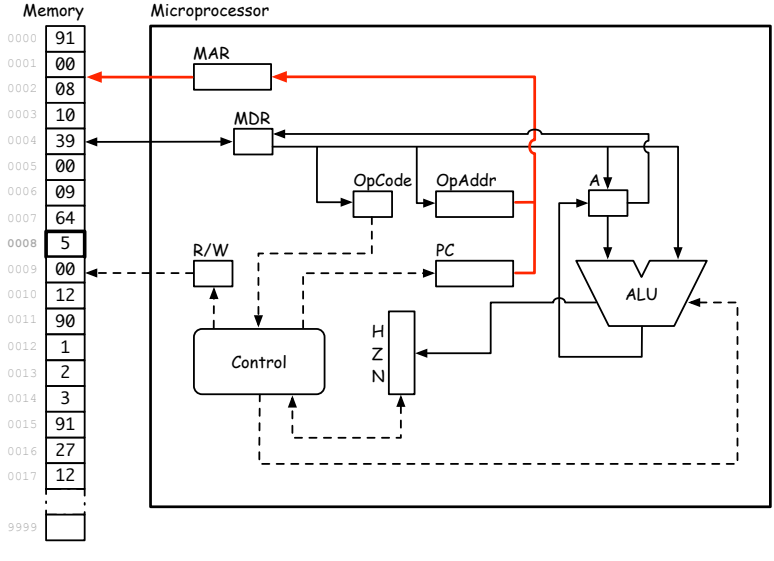
Préambule

Plan du module

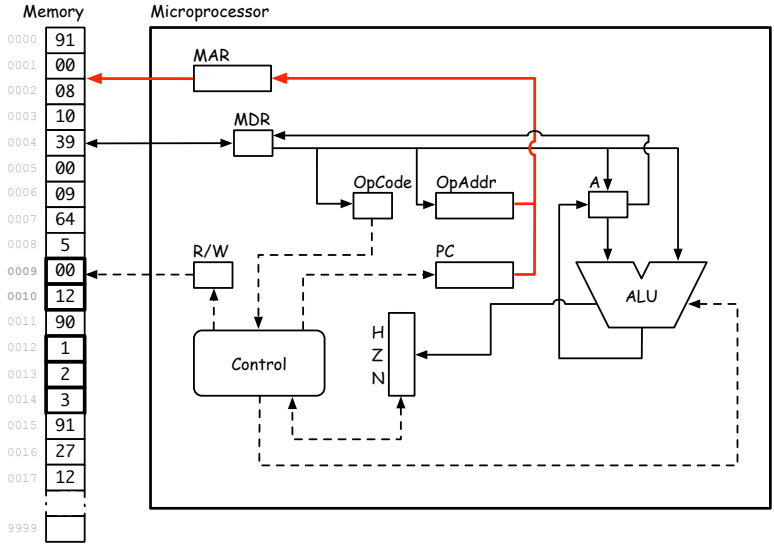
Plan

Rappel : Primitif vs référence et le TC-1101

```
int pos;
pos = 5;

int[] xs;
xs = new int[] {1, 2, 3};
```



- La variable **pos** est de type **int**, un type primitif, si **pos** désigne l'adresse **00 08**, alors la valeur **5** est sauvegardée à l'adresse **00 08**.



- La variable **xs** est de type **référence** vers un tableau d'entiers, si **xs** désigne l'adresse **00 09**, alors, la valeur des cellules **00 09** et **00 10**, est l'adresse où le tableau a été sauvegardé en mémoire, **00 12**. À l'adresse **00 12** se trouve le tableau, avec ses trois valeurs **1**, **2**, et **3**.

Diagrammes de mémoire

Diagramme de mémoire

Étant donné la déclaration suivante :

```
class Constant {
    String name;
    double value;
    Constant(String n, double v) {
        name = n;
        value = v;
    }
}
```

Dessinez le diagramme de mémoire qui correspond à ces énoncés :

```
Constant c;
c = new Constant("golden ratio", 1.61803399);
```

Diagramme de mémoire

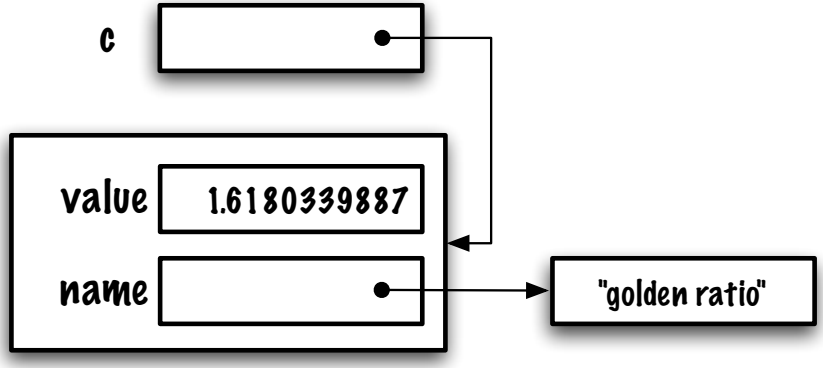
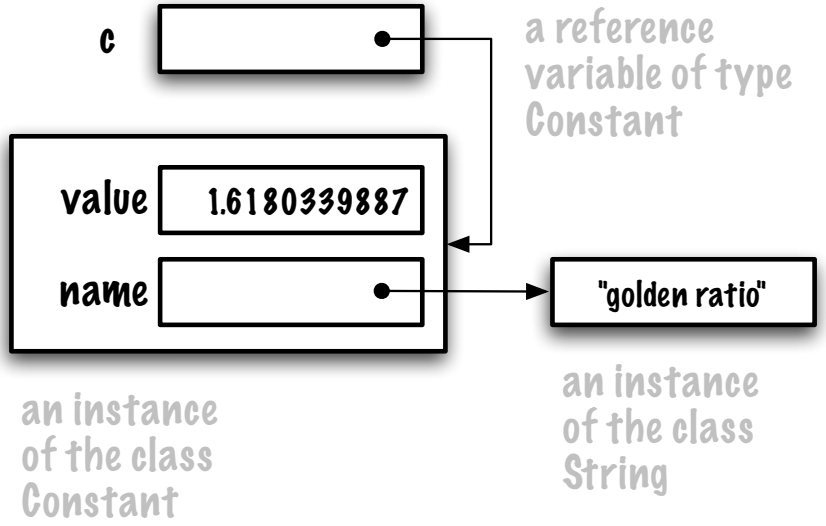


Diagramme de mémoire



Classe Integer

Pour les quelques exemples qui suivent, nous utiliserons une classe nommée **Integer** :

```
class Integer {
    int value;
}
```

Utilisation :

```
Integer a;
a = new Integer();
a.value = 33;
a.value++;
System.out.println("a.value = " + a.value);
```

On utilise la notation pointée afin d'accéder aux variables d'instance d'un objet.

Classe Integer

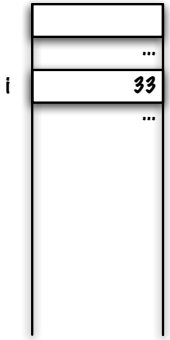
Ajout d'un constructeur

```
class Integer {
    int value;
    Integer(int v) {
        value = v;
    }
}
```

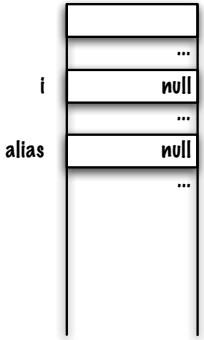
Utilisation :

```
Integer a;
a = new Integer(33);
```

Types primitifs et références



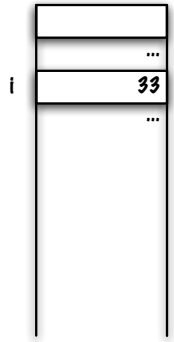

```
int i;
i = 33;
```



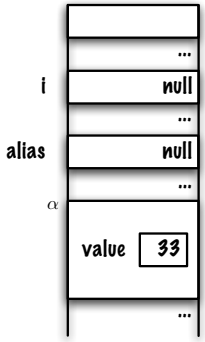

```
Integer i, alias;
i = new Integer(33);
alias = i;
```

À droite, lors de la **compilation**, une portion de la mémoire est réservée pour les variables références **i** et **alias**.

Types primitifs et références



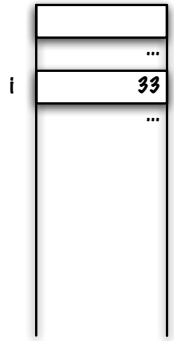

```
int i;
i = 33;
```



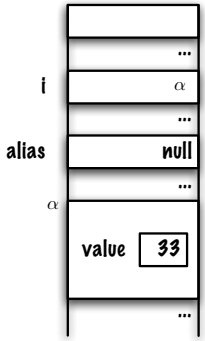

```
Integer i, alias;
i = new Integer(33);
alias = i;
```

Création d'un objet lors de l'**exécution** « **new Integer(33)** »

Types primitifs et références



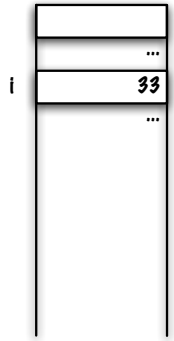

```
int i = 33;
```



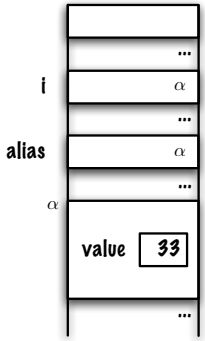

```
Integer i, alias;
i = new Integer(33);
alias = i;
```

Sauvegarder la **référence** de cet objet dans la variable référence **i**

Types primitifs et références



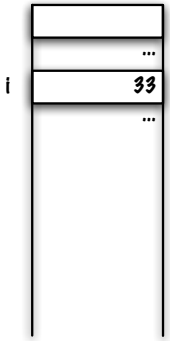

```
int i;
i = 33;
```



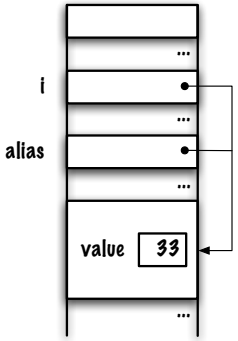

```
Integer i, alias;
i = new Integer(33);
alias = i;
```

Copier la **valeur** de variable référence **i** dans la variable **alias**

Types primitifs et références




```
int i;
i = 33;
```

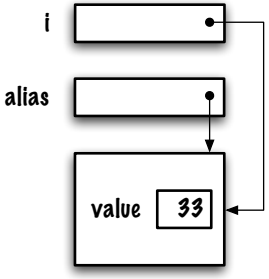



```
Integer i, alias;
i = new Integer(33);
alias = i;
```

i et **alias** désignent le même objet !

Types primitifs et références


```
int i;
i = 33;
```




```
Integer i, alias;
i = new Integer(33);
alias = i;
```

Diagramme de mémoire !

« Wrappers »

Classes enveloppantes (« wrappers »)

- Pour chaque **type primitif** il y a une **classe enveloppante** associée
- **Integer** est la classe enveloppante pour le type **int**
- Un **objet enveloppant** « entrepose » une valeur d'un type primitif dans un objet
- Nous les utiliserons avec les piles, files, listes et arbres
- Les classes enveloppantes possèdent aussi plusieurs méthodes pour faire la conversion de données,
  p.e. **Integer.parseInt("33")**

« Quiz »

Ces énoncés Java sont valide, **vrai** ou **faux** ?

```
Integer i;
i = 1;
```

- **S'ils sont valides**, quelles conclusions en tirez-vous ?
- 1 est une valeur d'un type primitif, **int**, mais **i** est une variable référence de type **Integer**
- Pour Java 1.2 ou 1.4, cet énoncé produira une **erreur de compilation**
- Cependant, pour Java 5, 6, 7 ou 8, cet énoncé est valide ! Pourquoi ?

Auto-boxing

Java 5, 6, 7 et 8 transforment **automagiquement** l'énoncé

```
Integer i = 1;
```

en celui-ci

```
Integer i = Integer.valueOf(1);
```

Ici, **valueOf** retourne la référence d'un objet de la classe **Integer** contenant la valeur de type primitif **int** spécifiée.*. C'est ce qu'on appelle **auto-boxing**

*https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Integer.html

Auto-unboxing

De même, l'énoncé **i = i + 5**

```
Integer i = 1;
i = i + 5;
```

sera transformé comme ceci

```
i = Integer.valueOf(i.intValue() + 5);
```

où la valeur de l'objet enveloppant désigné par **i** est extraite, **unboxed**, par l'appel **i.intValue()**

Boxing/unboxing

Primitive	Reference
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
bool	Boolean
char	Character

Les huit types primitifs on leur classe enveloppante (**wrapper**) associée.La conversion automatique d'un type primitif vers un type référence s'appelle **boxing**,et la conversion d'un type référence vers un type primitif s'appelle **unboxing**.

Dois-je m'en préoccuper ?

```
int s1;
s1 = 0;
for (int j=0; j<1000; j++) {
    s1 = s1 + 1;
}
```

9 093 nanosecondes

```
Integer s2;
s2 = 0;
for (int j=0; j<1000; j++) {
    s2 = s2 + 1;
}
```

335 023 nanosecondes

- Pourquoi ?

Du côté droit, **s2** est de type **Integer**, ainsi, l'énoncé

```
s2 = s2 + 1;
```

est réécrit (automatiquement) comme ceci

```
s2 = Integer.valueOf(s2.intValue() + 1);
```

Astuce de programmation : tests de performance

```
long start, stop, elapsed;

start = System.currentTimeMillis(); // start the clock

for (int j=0; j<10000000; j++) {
    s2 += 1; // stands for 's2 = s2 + 1'
}

stop = System.currentTimeMillis(); // stop the clock

elapsed = stop - start;
```

où **System.currentTimeMillis()** retourne le nombre de secondes qui se sont écoulées depuis minuit, 1er janvier, 1970 UTC (Coordinated Universal Time).
System.nanoTime() existe aussi !

Opérateurs

Opérateurs

Opérateurs de comparaison

Opérateurs de comparaison : types primitifs de données

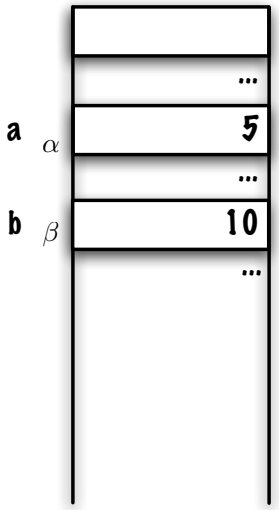
Les opérateurs de comparaison comparent les valeurs !

```java
int a = 5;
int b = 10;

if (a < b) {
  System.out.println("a < b");
} else if (a == b) {
  System.out.println("a == b");
} else {
  System.out.println("a > b");
}
```

Quel résultat sera imprimé en sortie ?

$\Rightarrow$ Affiche « $a < b$ »

```java
int a = 5;
int b = 10;

if (a < b) {
  System.out.println("a < b");
} else if (a == b) {
  System.out.println("a == b");
} else {
  System.out.println("a > b");
}
```

Opérateurs de comparaison : types primitifs et référence

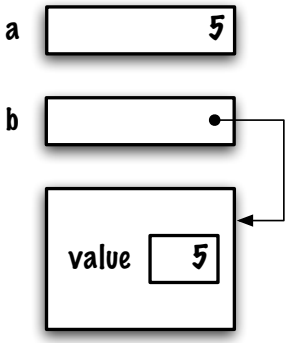
Quel est le résultat ?

```java
int a = 5;
Integer b = Integer.valueOf(5);
if (a < b) {
    System.out.println("a < b");
} else if (a == b) {
    System.out.println("a == b");
} else {
    System.out.println("a > b");
}
```

```
References.java:7: operator < cannot be applied to int,Integer
       if (a < b)
             ^
References.java:9: operator == cannot be applied to int,Integer
       else if (a == b)
                  ^
2 errors
```

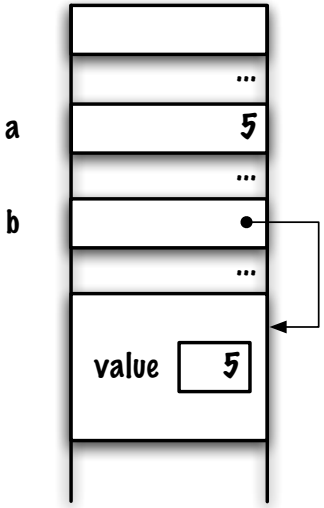
```java
int a = 5;
Integer b = Integer.valueOf(5);

if (a < b) {
    System.out.println("a < b");
} else if (a == b) {
    System.out.println("a == b");
} else {
    System.out.println("a > b");
}
```

```
References.java:7: operator < cannot be applied to int,Integer
        if (a < b)
              ^
References.java:9: operator == cannot be applied to int,Integer
        else if (a == b)
                   ^
2 errors
```



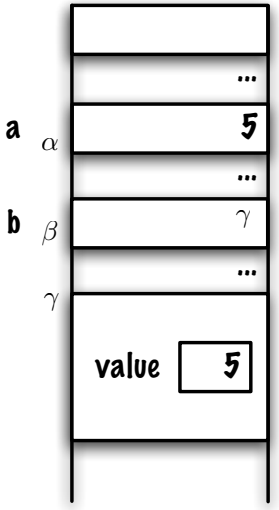
```java
int a = 5;
Integer b = Integer.valueOf(5);

if (a < b) {
    System.out.println("a < b");
} else if (a == b) {
    System.out.println("a == b");
} else {
    System.out.println("a > b");
}
```

```
References.java:7: operator < cannot be applied to int,Integer
        if (a < b)
               ^
References.java:9: operator == cannot be applied to int,Integer
        else if (a == b)
                    ^
2 errors
```



```java
int a = 5;
Integer b = Integer.valueOf(5);

if (a < b) {
    System.out.println("a < b");
} else if (a == b) {
    System.out.println("a == b");
} else {
    System.out.println("a > b");
}
```

```
References.java:7: operator < cannot be applied to int,Integer
        if (a < b)
               ^
References.java:9: operator == cannot be applied to int,Integer
        else if (a == b)
                    ^
2 errors
```



Remarques

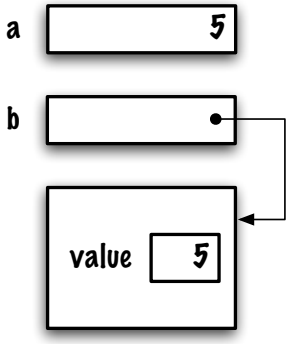
- Ces messages d'erreurs sont produits par les compilateurs pré-1.5
- Pour 1.5 et $+$, l'autoboxing masquera (possiblement) le « problème »
- Pour obtenir le même comportement pour les deux environnements, utilisons notre propre classe enveloppante, **MyInteger**

Classe MyInteger

```
class MyInteger {
    int value;
    MyInteger(int v) {
        value = v;
    }
}
```

```java
int a = 5;
MyInteger b = new MyInteger(5);

if (a < b) {
    System.out.println("a < b");
} else if (a == b) {
    System.out.println("a == b");
} else {
    System.out.println("a > b");
}
```

- Corrigez ces énoncés !



Solution

```
int a = 5;
MyInteger b = new MyInteger(5);

if (a < b.value) {
    System.out.println("a is less than b");
} else if (a == b.value) {
    System.out.println("a equals b");
} else {
    System.out.println("a is greater than b");
}
```

⇒ Prints « **a equals b** »

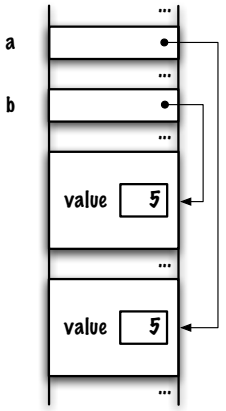
Opérateurs de comparaison et types références

Que se passera-t-il et pourquoi ?

```
MyInteger a = new MyInteger(5);
MyInteger b = new MyInteger(5);

if (a == b) {
    System.out.println("a equals b");
} else {
    System.out.println("a does not equal b");
}
```

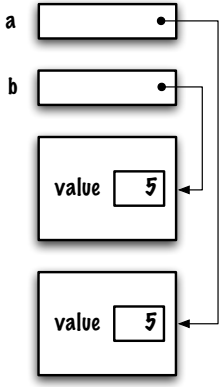
⇒ Affiche « **a does not equal b** »

```
MyInteger a = new MyInteger(5);
MyInteger b = new MyInteger(5);

if (a == b) {
  System.out.println("a equals b");
} else {
  System.out.println("a does not equals b");
}
```



⇒ Affiche « **a does not equal b** »

```
MyInteger a = new MyInteger(5);
MyInteger b = new MyInteger(5);

if (a == b) {
  System.out.println("a equals b");
} else {
  System.out.println("a does not equals b");
}
```



⇒ Affiche « **a does not equal b** »

Solution

```
MyInteger a = new MyInteger(5);
MyInteger b = new MyInteger(5);

if (a.equals(b)) {
  System.out.println("a equals b");
} else {
  System.out.println("a does not b");
}
```

où la méthode **equals** aurait été définie comme ceci

```
public boolean equals(MyInteger other) {
    boolean answer = false;
    if (value == other.value) {
        answer = true;
    }
    returns answer;
}
```

$\Rightarrow$ Affiche « **a equals b** »

Quel est le résultat ?

```
MyInteger a = new MyInteger(5);
MyInteger b = a;

if (a == b) {
  System.out.println("a == b");
} else {
  System.out.println("a != b");
}
```

⇒ Affiche « **a == b** », pourquoi ? parce que les références **a** et **b** désigne le même objet, la même instance, autrement dit, les deux adresses mémoires sont identiques ; on dit que **a** et **b** sont des alias.

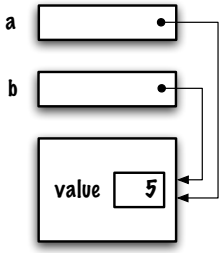
Opérateurs de comparaison et types références

Quel sera le résultat ?

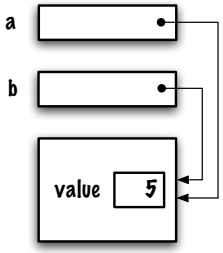
```
MyInteger a = new MyInteger(5);
MyInteger b = a;

if (a.equals(b)) {
  System.out.println("a equals b");
} else {
  System.out.println("a does not equal b");
}
```

⇒ Affiche « **a equals b** »

```
MyInteger a = new MyInteger (5);
MyInteger b = a;

if (a == b) {
  System.out.println("a == b");
} else {
  System.out.println("a != b");
}
```



```java
MyInteger a = new MyInteger(5);
MyInteger b = a;

if (a.equals(b)) {
  System.out.println("a == b");
} else {
  System.out.println("a != b");
}
```



Opérateurs de comparaison et types références

Quel sera le résultat ?

```
MyInteger a = new MyInteger(5);
MyInteger b = new MyInteger(10);

if (a < b) {
  System.out.println("a equals b");
} else {
  System.out.println("a does not equal b");
}
```

```
Less.java:14: operator < cannot be applied to MyInteger,MyInteger
        if (a < b) {
               ^
1 error
```

Remarques

- Pour comparer le contenu des objets désignés, « équivalence de contenu » ou « équivalence logique », on utilise **equals**[†]
- Pour savoir si deux variables références **désignent le même objet**, on utilise les opérateurs de comparaison **'=='** et **' !='**

```
if (a.equals(b)) { ... }
```

vs

```
if (a == b) { ... }
```

[†] À suivre...

Exercices

Comparez les objets deux à deux en utilisant soit **equals** ou **==**, vous pourriez être surpris.

```
String a = new String("Hello");
String b = new String("Hello");
int c[] = { 1, 2, 3 };
int d[] = { 1, 2, 3 };
String e = "Hello";
String f = "Hello";
String g = f + "";
```

En particulier, essayez **a == b** et **e == f**.

Appel de méthode

Appel de méthode

Définitions

Définition : arité

L'**arité** d'une méthode est le nombre de paramètres ; une méthode possède aucun, un ou plusieurs paramètres.

```
MyInteger() {
  value = 0;
}
MyInteger(int v) {
  value = v;
}
int sum(int a, int b) {
  return a + b;
}
```

Définition : paramètre formel

Un **paramètre formel** est une variable qui fait partie de la signature de la méthode ; elle peut être vue comme une variable locale du corps de la méthode.

```
int sum(int a, int b) {
    return a + b;
}
```

⇒ **a** et **b** sont les paramètres formels de **sum**.

Définition : paramètre actuel

Le **paramètre actuel** est la variable qui est utilisée lors de l'appel de méthode et fournit la valeur initiale au **paramètre formel**.

```
int sum(int a, int b) {
    return a + b;
}
...
int midTerm, finalExam, total;
total = sum(midTerm, finalExam);
```

midTerm et **finalExam** sont les paramètres actuels de l'appel à la méthode **sum**, lors de l'appel la **valeur** du paramètre actuel est copiée à l'emplacement du paramètre formel.

Concept : appel-par-valeur

En Java, lors d'un appel de méthode, la **valeur** d'un paramètre actuel **copiée** à l'emplacement du paramètre formel.
Lors d'un appel de méthode :

- l'exécution de la méthode appelante est interrompue
- il y a création d'un bloc de mémoire de travail [‡]
  (qui contient les paramètres formels et variables locales)
- **les valeurs des paramètres actuels sont affectées aux paramètres formels**
- le corps de la méthode est exécuté
- la valeur de retour est sauvée
- (le bloc de mémoire de travail est détruit)
- l'exécution de la méthode appelante se poursuit avec l'instruction qui suit l'appel de méthode

[‡] bloc d'activation

Appel de méthode

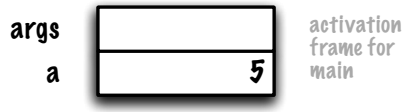
Types primitifs

```java
public class Test {

  public static void increment(int a) {
      a = a + 1;
  }

  public static void main(String[] args) {
      int a = 5;
      System.out.println("before: " + a);
      increment(a);
      System.out.println("after: " + a);
  }
}
```

Quel sera le résultat ?

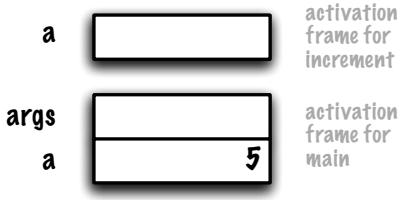
```
before: 5
after: 5
```

```
   public static void increment(int a) {
       a = a + 1;
   }
   public static void main(String[] args) {
>      int a = 5;
       increment(a);
   }
```



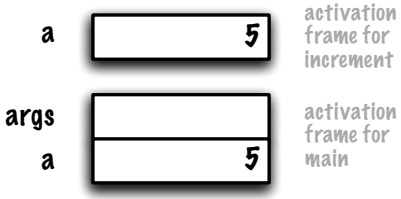
Chaque **appel de méthode** possède sa propre **mémoire de travail** (bloc d'activation), afin de sauvegarder les paramètres et variables locales à cet appel (ici, **args** et **a**)

```
   public static void increment(int a) {
       a = a + 1;
   }
   public static void main(String[] args) {
       int a = 5;
>      increment(a);
   }
```



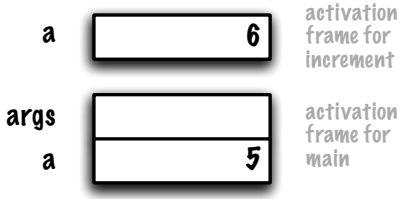
Lors de l'appel à la méthode **increment** un nouveau bloc de mémoire de travail est créé

```
   public static void increment(int a) {
       a = a + 1;
   }
   public static void main(String[] args) {
       int a = 5;
>      increment(a);
   }
```



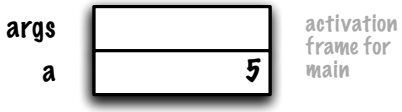
La valeur de chaque **paramètre actuel** est copié à l'emplacement du **paramètre formel** correspondant

```
  public static void increment(int a) {
>     a = a + 1;
  }
  public static void main(String[] args) {
      int a = 5;
      increment(a);
  }
```



L'exécution de l'énoncé $\mathbf{a = a + 1}$ change la valeur du paramètre formel **a**, un emplacement mémoire distinct de celui de la variable locale **a** de la méthode **main**

```
   public static void increment(int a) {
       a = a + 1;
   }
   public static void main(String[] args) {
       int a = 5;
       increment(a);
> }
```



Le contrôle retourne à la méthode **main**, le bloc mémoire de travail pour la méthode **increment** est détruit

Appel de méthode

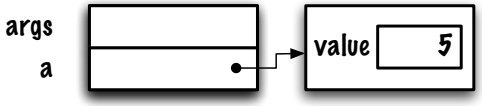
Variables références

Références et appels de méthodes

```
class MyInteger {
    int value;
    MyInteger(int v) {
        value = v;
    }
}
class Test {
    public static void increment(MyInteger a) {
        a.value++;
    }
    public static void main(String[] args) {
        MyInteger a = new MyInteger (5);
        System.out.println("before: " + a.value);
        increment(a);
        System.out.println("after: " + a.value);
    }
}
```

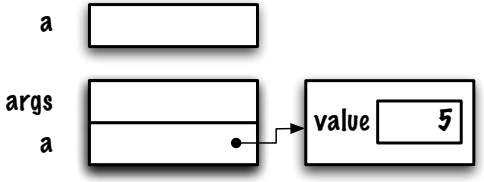
Quel sera le résultat ?

```
  static void increment(MyInteger a) {
      a.value++;
  }
  public static void main(String[] args) {
>     MyInteger a = new MyInteger(5);
      increment(a);
  }
```



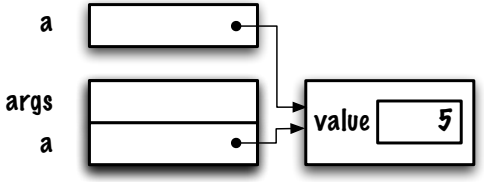
La variable locale **a** de la méthode **main** est une référence vers un objet de la classe **MyInteger**

```
static void increment(MyInteger a) {
    a.value++;
}
public static void main(String[] args) {
    MyInteger a = new MyInteger(5);
>   increment(a);
}
```



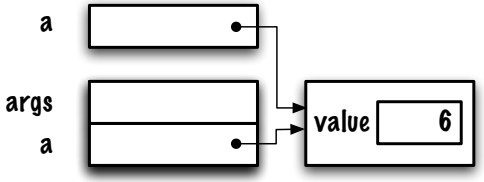
Appel à **increment**, création d'un bloc d'activation

```
static void increment(MyInteger a) {
    a.value++;
}
public static void main(String[] args) {
    MyInteger a = new MyInteger(5);
>   increment(a);
}
```



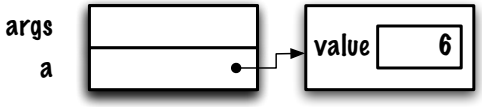
Copier la **valeur** du **paramètre actuel** à l'emplacement du **paramètre formel**

```
 static void increment(MyInteger a) {
>    a.value++;
 }
 public static void main(String[] args) {
     MyInteger a = new MyInteger(5);
     increment(a);
 }
```



Exécuter **a.value++**

```
  static void increment(MyInteger a) {
      a.value++;
  }
  public static void main(String[] args) {
      MyInteger a = new MyInteger(5);
      increment(a);
>}
```



Retourne le contrôle à la méthode **main**

```
public class TestSwapArrayInt {
        public static void swap(int[] xs) {
            int[] ys;
            ys = new int[2];
            ys[0] = xs[1];
            ys[1] = xs[0];
            xs = ys;
        }
        public static void main(String[] args) {
            int[] xs;
            xs = new int[2];
            xs[0] = 15;
            xs[1] = 21;
            swap(xs);
            System.out.println(xs[0]);
            System.out.println(xs[1]);
        }
}
```

Portée

Portée

Définitions

Définition : portée

*La **portée** d'une déclaration est la région du programme à l'intérieur de laquelle on peut référencer l'entité déclarée par la déclaration à l'aide d'un nom simple*

The Java Language Specification,
Third Edition, Addison Wesley, p. 117.

Définition : portée

*The **scope** of a declaration is the region of the program within which the entity declared by the declaration can be referred to using a simple name*

The Java Language Specification,
Third Edition, Addison Wesley, p. 117.

Définition : portée d'une variable locale en Java

*La **portée de la déclaration d'une variable locale** dans un bloc d'énoncés est le reste du bloc dans lequel cette déclaration apparaît*

The Java Language Specification,
Third Edition, Addison Wesley, p. 118.

$\Rightarrow$ A.K.A. portée statique ou lexicale

Définition : portée d'une variable locale en Java

*The **scope of a local variable declaration** in a block is the rest of the block in which the declaration appears, starting with its own initializer and including any further declarators to the right in the local variable declaration statement*

The Java Language Specification,
Third Edition, Addison Wesley, p. 118.

$\Rightarrow$ A.K.A. static or lexical scope

Définition : portée d'un paramètre en Java

*La **portée d'un paramètre** d'une méthode ou d'un constructeur est corps en entier de la méthode ou du constructeur*

The Java Language Specification,
Third Edition, Addison Wesley, p. 118.

$\Rightarrow$ A.K.A. portée statique ou lexicale

Portée

Exemples

```java
public class Test {
    public static void display() {
        System.out.println("a = " + a);
    }
    public static void main(String[] args) {
        int a;
        a = 9; // valid access, within the same block
        if (a < 10) {
            a = a + 1; // another valid access
        }
        display();
    }
}
```

S'agit-il d'un programme Java valide ?

```java
public class Test {
    public static void main(String[] args) {
        System.out.println(sum);
        for (int i=1; i<10; i++) {
            System.out.println(i);
        }x
        int sum = 0;
        for (int i=1; i<10; i++) {
            sum += i;
        }
    }
}
```

S'agit-il d'un programme Java valide ?

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

        for (int i=1; i<10; i++) {
            System.out.println(i);
        }
        int sum = 0;
        for (int i=1; i<10; i++) {
            sum += i;
        }
    }
}
```

S'agit-il d'un programme Java valide ?

Gestion de la mémoire

Qu'arrive-t-il aux objets lorsqu'**aucune référence ne les désignent** ? Ici, qu'advient-il de l'objet contenant la valeur 99 ?

```
MyInteger a = new MyInteger(7);
MyInteger b = new MyInteger(99);
b = a;
```

- La JVM récupérera l'espace mémoire associé !
- Ce processus s'appelle **garbage collection**
- Certains langages de programmation ne gèrent pas automatique les allocations et désallocations de mémoire

Java n'est cependant pas immunisé contre les fuites de mémoire, à suivre...

Prologue

Résumé

- Les opérateurs de comparaisons comparent les valeurs des variables
  - En particulier, si **a** et **b** sont des variables références, alors **a == b** returne **true** ssi **a** et **b** désigne le même objet.
- Les appels de méthodes sont **par valeur** en Java

Prochain module

- **Programmation orientée objet**

References I


E. B. Koffman and Wolfgang P. A. T.
Data Structures : Abstraction and Design Using Java.
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

École de **science informatique** et de génie électrique (SIGE)

Université d'Ottawa