ITI 1521. Introduction à l'informatique II †

Marcel Turcotte
(contributions de R. Holte)

École de science informatique et de génie électrique
Université d'Ottawa

Version du 11 janvier 2012

†. Pensez-y, n'imprimez ces notes de cours que si c'est nécessaire!

Revue

Objectifs :

1. Discuter plusieurs concepts liés aux types de données
2. Comprendre les implications des différences entre les types primitifs et les types références
3. Revoir le concept d'appel par valeur
4. Connaître le concept de portée

Lectures :

- Pages 597–631 de E. Koffman and P. Wolfgang.

Plan

1. Que sont les variables et le types de données
2. Types primitif et référence
3. Opérateurs de comparaison
4. Auto-boxing/auto-unboxing
5. Passage de paramètres
6. Portée
7. Gestion de la mémoire

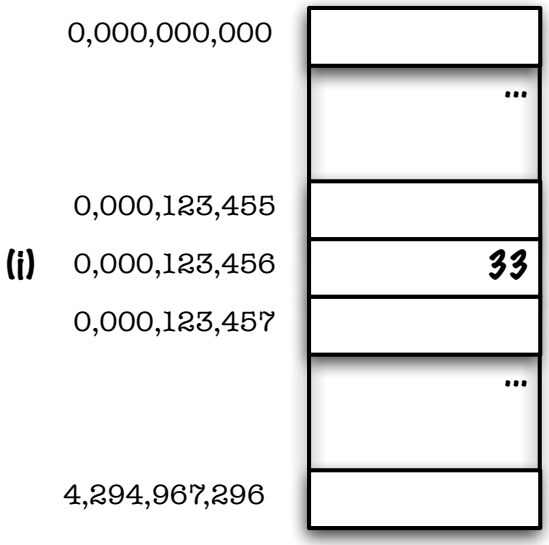
Variables

Qu'est-ce qu'une variable ?

Variables

Qu'est-ce qu'une variable ?

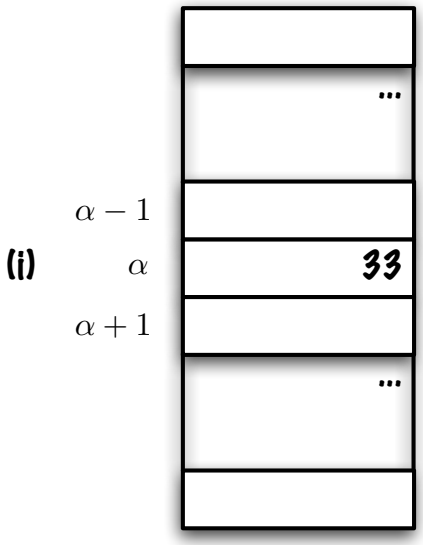
- C'est un emplacement de la mémoire, pour une **valeur**, auquel on réfère à l'aide d'une **étiquette**, dans les langages de haut niveau :




```
byte i = 33;
```

Variables

J'utiliserai des lettres grecques pour désigner les emplacements (adresses) en mémoire puisqu'en Java on ne connais pas l'emplacement des objets et on ne devrait pas s'en préoccuper.




```
byte i = 33;
```

Types de données

À quoi servent les types de données ?

Types de données

À quoi servent les types de données ?

- En effet, le type d'une variable indique au compilateur, et à la JVM, la portion de la mémoire qui doit être réservée pour cette variable (**implémentation**) :

```
double formula;          // 8 octets
char c;                  // 2 octets
```

Types de données

À quoi servent les types de données ?

- En effet, le type d'une variable indique au compilateur, et à la JVM, la portion de la mémoire qui doit être réservée pour cette variable (**implémentation**) :

```
double formula;          // 8 octets
char c;                  // 2 octets
```

- Mais aussi ?

Types de données

À quoi servent les types de données ?

- En effet, le type d'une variable indique au compilateur, et à la JVM, la portion de la mémoire qui doit être réservée pour cette variable (**implémentation**) :

```
double formula;          // 8 octets
char c;                  // 2 octets
```

- Mais aussi ? le type **détermine les opérations qui sont permises** sur cette variable (**sémantique**), quelles données sont compatibles

Types de données

À quoi servent les types de données ?

- En effet, le type d'une variable indique au compilateur, et à la JVM, la portion de la mémoire qui doit être réservée pour cette variable (**implémentation**) :

```
double formula;          // 8 octets
char c;                  // 2 octets
```

- Mais aussi ? le type **détermine les opérations qui sont permises** sur cette variable (**sémantique**), quelles données sont compatibles

```
c = flag * formula;
```

Ainsi, l'énoncé ci-haut produira une erreur lors de la compilation du programme ; les types de données sont donc utiles afin de détecter les erreurs de programmation tôt

Types de données (suite)

- On distingue les **types concrets de données** et les **types abstraits de données**

Types de données (suite)

- On distingue les **types concrets de données** et les **types abstraits de données**
- Les types concrets de données spécifient 1) les opérations permises ainsi que 2) la représentation des données

Types de données (suite)

- On distingue les **types concrets de données** et les **types abstraits de données**
- Les types concrets de données spécifient 1) les opérations permises ainsi que 2) la représentation des données
- **Les types abstraits de données ne spécifient que les opérations permises**

Types de données en Java

- Les types primitifs sont :

Types de données en Java

- Les types primitifs sont :
  - nombres (int, long, float, double), les caractères (char, mais pas les chaînes) et les booléens

Types de données en Java

- Les types primitifs sont :
  - nombres (int, long, float, double), les caractères (char, mais pas les chaînes) et les booléens
  - **la valeur d'une variable d'un type primitif se trouve à l'adresse désignée par l'étiquette (identificateur)**

Types de données en Java

- Les types primitifs sont :
  - nombres (int, long, float, double), les caractères (char, mais pas les chaînes) et les booléens
  - **la valeur d'une variable d'un type primitif se trouve à l'adresse désignée par l'étiquette (identificateur)**
- Références :

Types de données en Java

- Les types primitifs sont :
  - nombres (int, long, float, double), les caractères (char, mais pas les chaînes) et les booléens
  - **la valeur d'une variable d'un type primitif se trouve à l'adresse désignée par l'étiquette (identificateur)**
- Références :
  - Prédéfines :

Types de données en Java

- Les types primitifs sont :
  - nombres (int, long, float, double), les caractères (char, mais pas les chaînes) et les booléens
  - **la valeur d'une variable d'un type primitif se trouve à l'adresse désignée par l'étiquette (identificateur)**
- Références :
  - Prédéfines :
    - Arrays (tableaux)

Types de données en Java

- Les types primitifs sont :
  - nombres (int, long, float, double), les caractères (char, mais pas les chaînes) et les booléens
  - **la valeur d'une variable d'un type primitif se trouve à l'adresse désignée par l'étiquette (identificateur)**
- Références :
  - Prédéfines :
    - Arrays (tableaux)
    - Strings (chaînes de caractères)

Types de données en Java

- Les types primitifs sont :
  - nombres (int, long, float, double), les caractères (char, mais pas les chaînes) et les booléens
  - **la valeur d'une variable d'un type primitif se trouve à l'adresse désignée par l'étiquette (identificateur)**
- Références :
  - Prédéfines :
    - Arrays (tableaux)
    - Strings (chaînes de caractères)
  - Les types définis par l'usager,

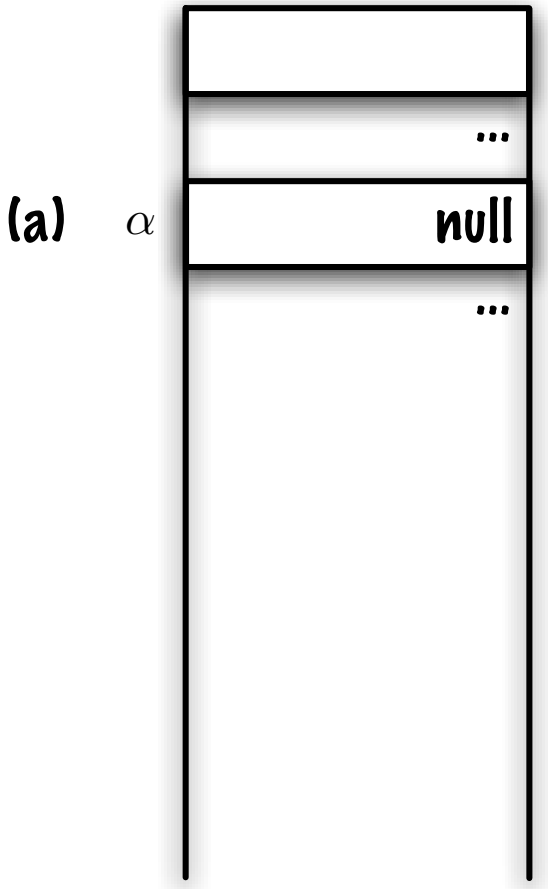
Types de données en Java

- Les types primitifs sont :
  - nombres (int, long, float, double), les caractères (char, mais pas les chaînes) et les booléens
  - **la valeur d'une variable d'un type primitif se trouve à l'adresse désignée par l'étiquette (identificateur)**
- Références :
  - Prédéfines :
    - Arrays (tableaux)
    - Strings (chaînes de caractères)
  - Les types définis par l'usager,

Types de données en Java

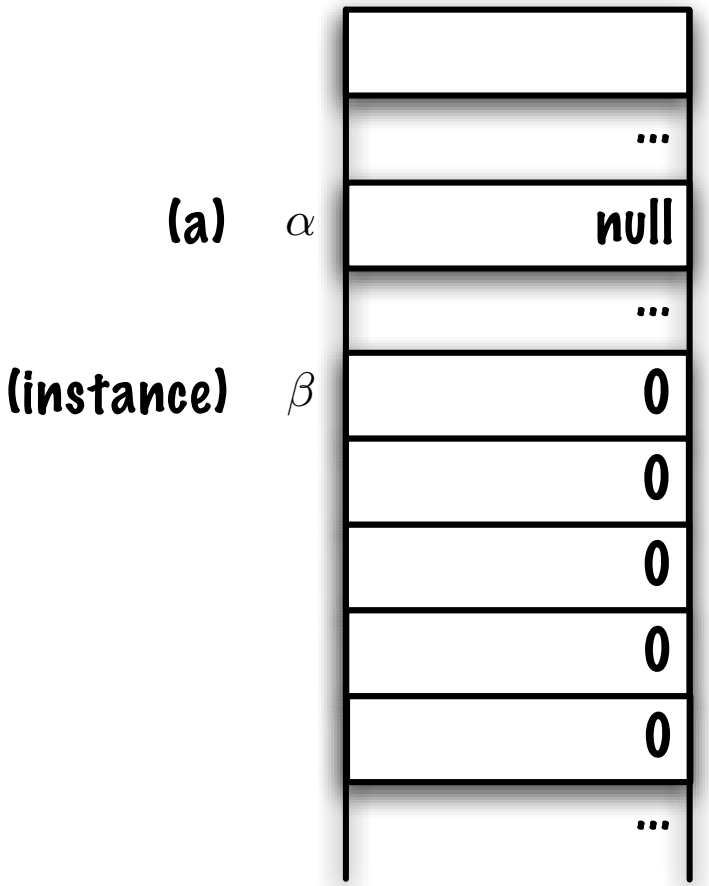
- Les types primitifs sont :
  - nombres (int, long, float, double), les caractères (char, mais pas les chaînes) et les booléens
  - **la valeur d'une variable d'un type primitif se trouve à l'adresse désignée par l'étiquette (identificateur)**
- Références :
  - Prédéfines :
    - Arrays (tableaux)
    - Strings (chaînes de caractères)
  - Les types définis par l'usager, référence vers un objet
  - **La valeur d'une variable de type référence est l'adresse de l'emplacement mémoire de l'objet désigné par la variable — on dit que la variable pointe, désigne ou référence l'objet**

```
> int a [];
  a = new int[5];
```



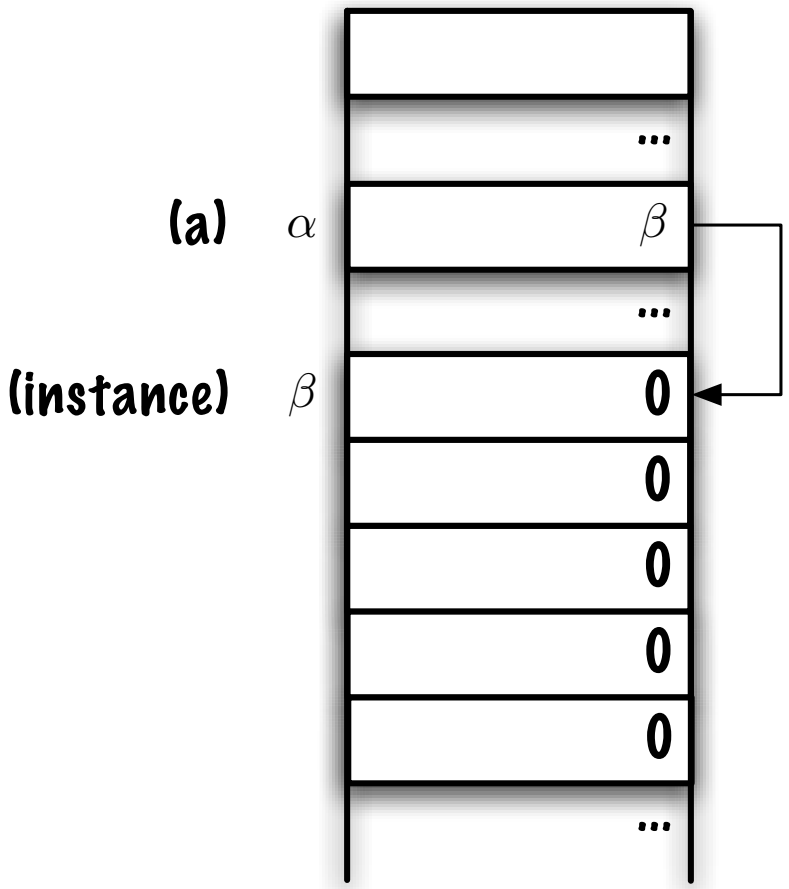
La déclaration d'une variable de type référence **ne crée pas l'objet (instance)**, le compilateur réservera un espace suffisant pour contenir la référence (pointeur), **null** est un littéral qui signifie : ne désigne aucun objet.

```
   int a[];
> a = new int[ 5 ];
```



La création d'un objet, **new int[5]**, réserve une portion de la mémoire pour 5 entiers (et la gestion interne — *housekeeping*). Chaque cellule du tableau se comporte comme une variable de type **int** et reçoit la valeur initiale 0.

```
    int a[];
>   a = new int[ 5 ];
```



Finalement, la référence du nouvel objet est sauvegardée à l'adresse désignée par l'étiquette **a**.

Diagramme de mémoire

Puisqu'on ne connaît pas l'emplacement des objets en mémoire (et qu'on ne devrait pas s'en préoccuper), nous utiliserons des diagrammes de mémoire (image la plus à droite)

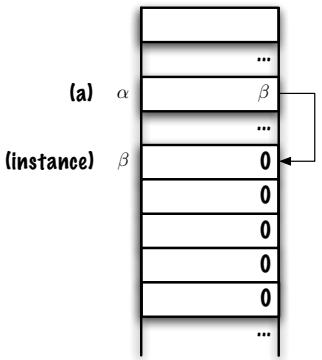


Diagramme de mémoire

Puisqu'on ne connaît pas l'emplacement des objets en mémoire (et qu'on ne devrait pas s'en préoccuper), nous utiliserons des diagrammes de mémoire (image la plus à droite)

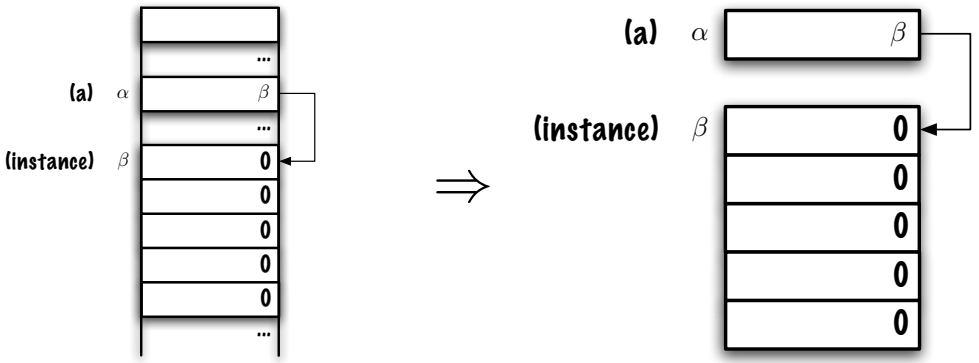


Diagramme de mémoire

Puisqu'on ne connaît pas l'emplacement des objets en mémoire (et qu'on ne devrait pas s'en préoccuper), nous utiliserons des diagrammes de mémoire (image la plus à droite)

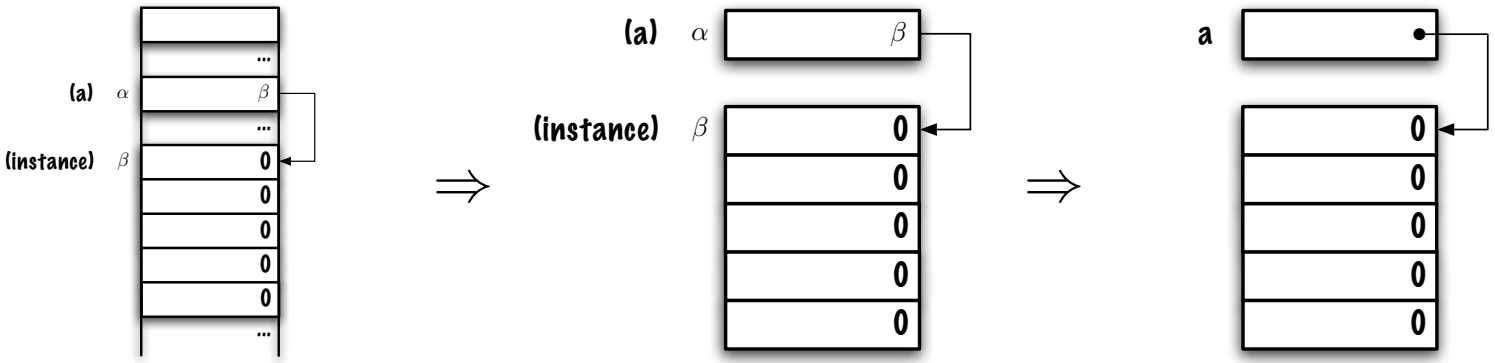


Diagramme de mémoire

Consignes pour vos diagrammes de mémoire :

- Une boîte pour chaque variable référence et une flèche vers l'objet désigné
- Une boîte pour chaque variable de type primitif et sa valeur dans la boîte même

```
int[] a;
a = new int[ 5 ];
```

Diagramme de mémoire

Consignes pour vos diagrammes de mémoire :

- Une boîte pour chaque variable référence et une flèche vers l'objet désigné
- Une boîte pour chaque variable de type primitif et sa valeur dans la boîte même

```
int[] a;
a = new int[ 5 ];
```

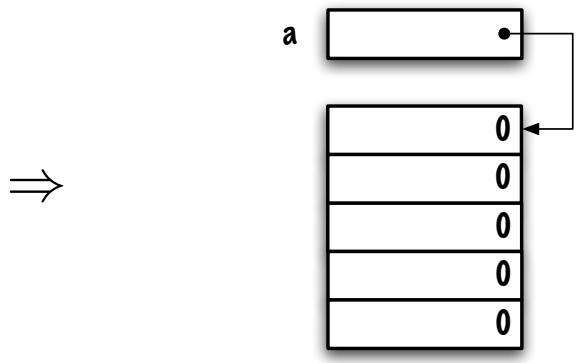


Diagramme de mémoire

Étant donné la déclaration suivante :

```
public class Constant {
    private final String name;
    private final double value;
    public Constant( String name, double value ) {
        this.name = name;
        this.value = value;
    }
}
```

Diagramme de mémoire

Étant donné la déclaration suivante :

```
public class Constant {
    private final String name;
    private final double value;
    public Constant( String name, double value ) {
        this.name = name;
        this.value = value;
    }
}
```

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

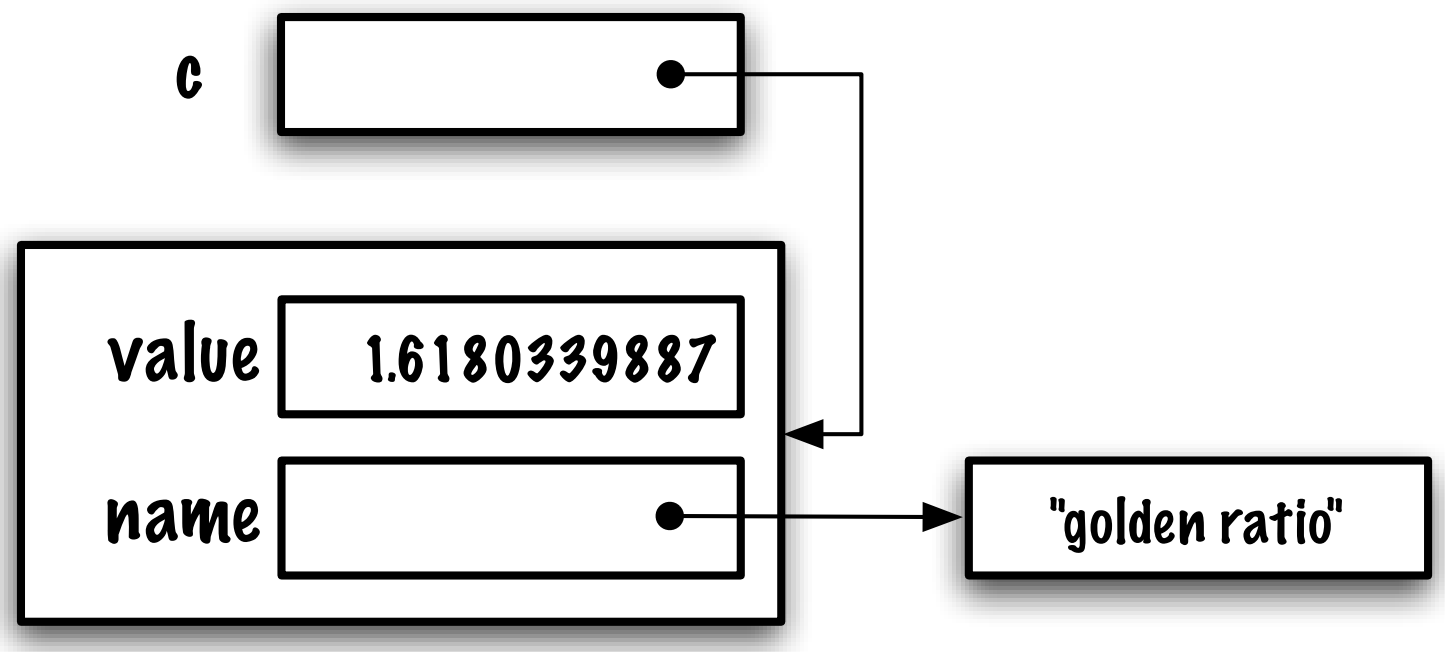
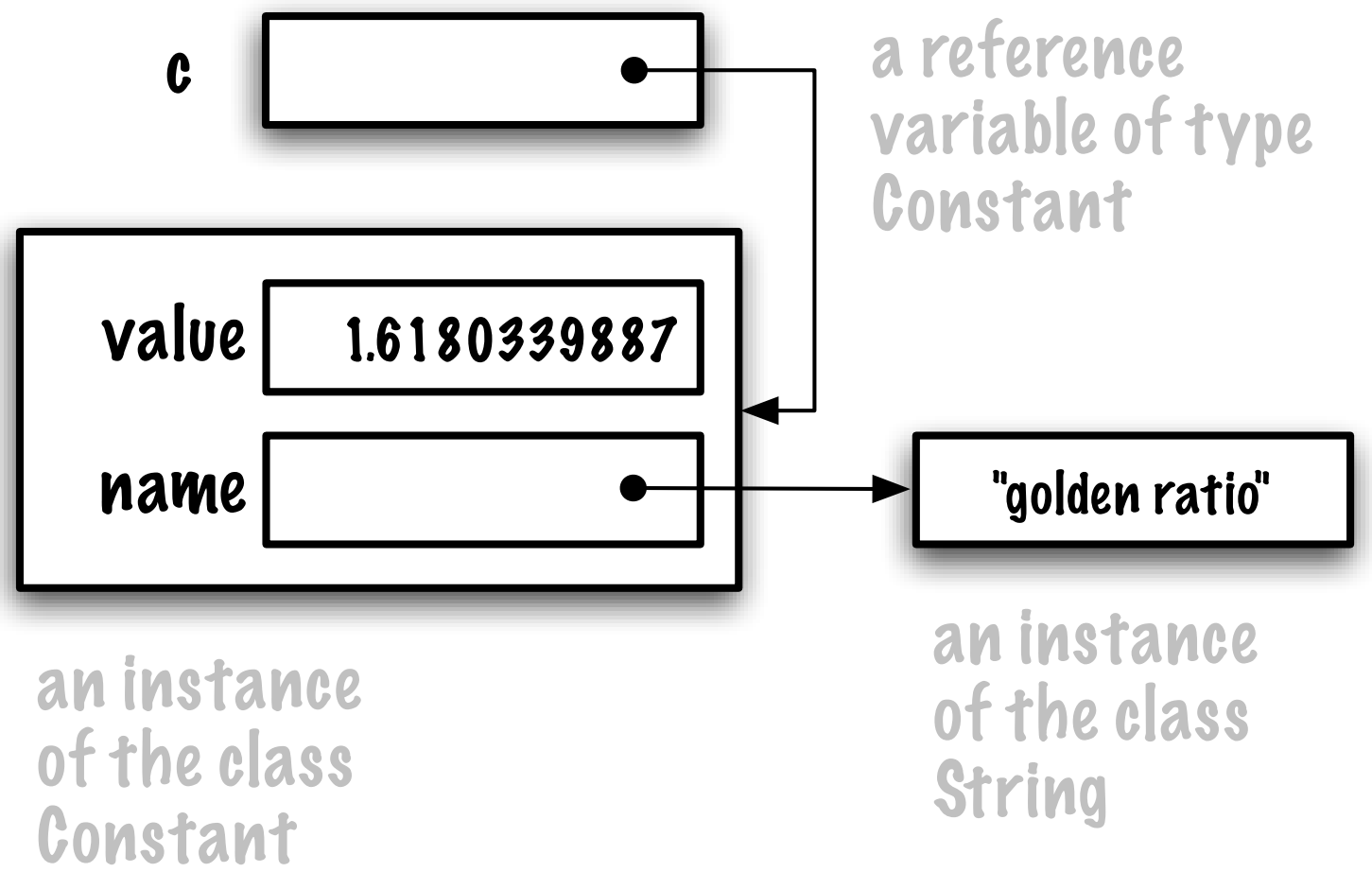


Diagramme de mémoire



Rappel

- « **Variables have types** »
- « **Objects have classes** »

Classe Integer

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

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

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

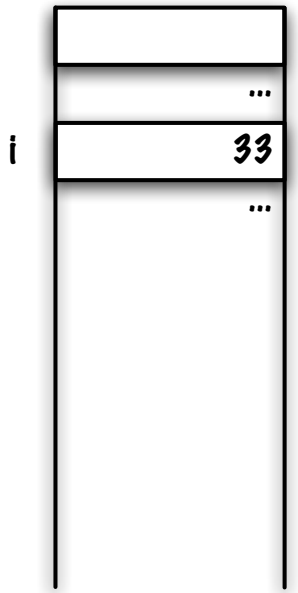
Ajout d'un constructeur

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

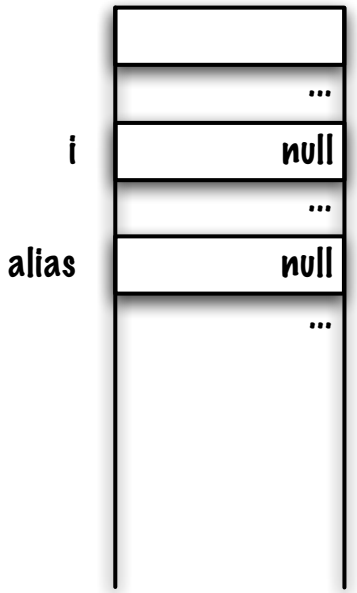
Utilisation :

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

Types primitifs et références



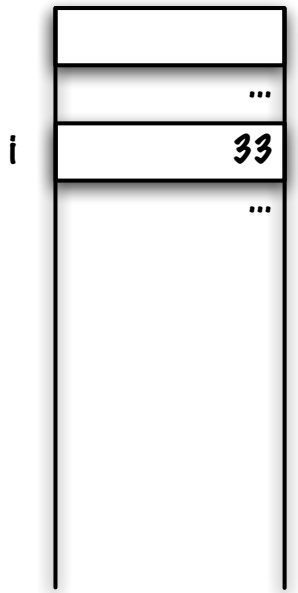

```
int i = 33;
```



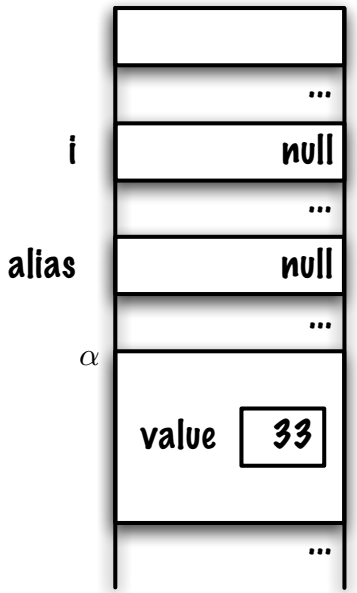

```
Integer i, alias;
i = new Integer( 33 );
alias = i;
```

Lors de la **compilation**, une portion de la mémoire est réservée pour la référence

Types primitifs et références



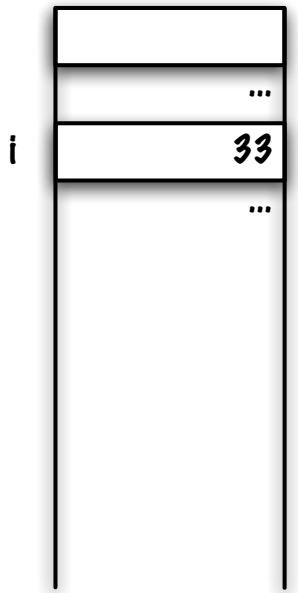

```
int i = 33;
```



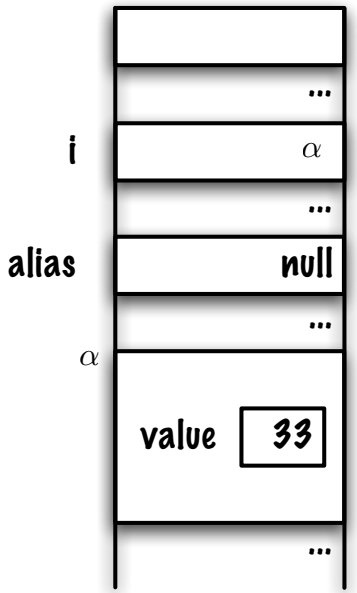

```
Integer i, alias;
i = new Integer( 33 );
alias = i;
```

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

Types primitifs et références



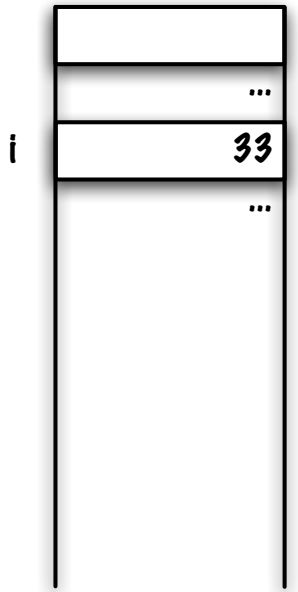

```
int i = 33;
```



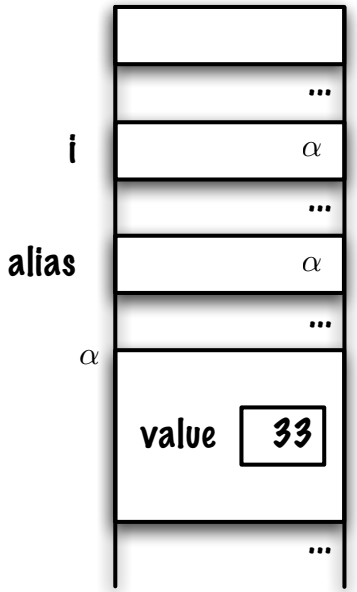

```
Integer i, alias;
i = new Integer( 33 );
alias = i;
```

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

Types primitifs et références



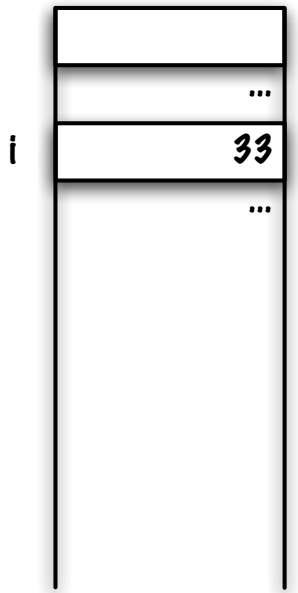

```
int i = 33;
```



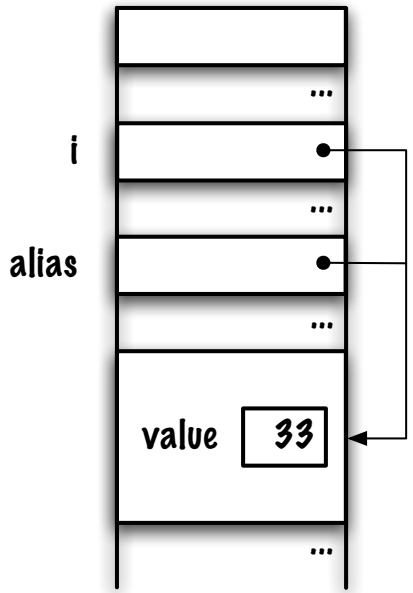

```
Integer i, alias;
i = new Integer( 33 );
alias = i;
```

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

Types primitifs et références




```
int i = 33;
```

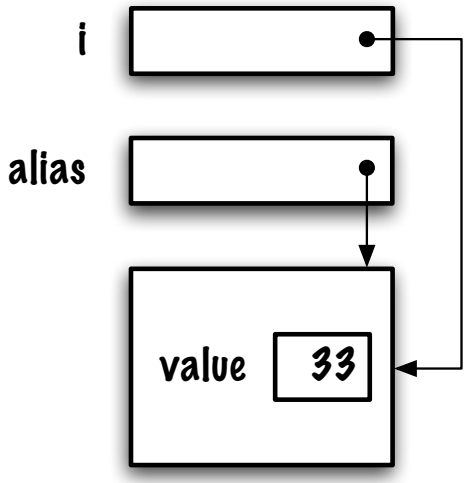



```
Integer i, alias;
i = new Integer( 33 );
alias = i;
```

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

Types primitifs et références


```
int i = 33;
```




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

Diagramme de mémoire

Classes enveloppantes (wrappers)

- Pour chaque **type primitif** il y a une **classe enveloppante** associée

Classes enveloppantes (wrappers)

- Pour chaque **type primitif** il y a une **classe enveloppante** associée
- **Integer** est la classe enveloppante pour le type **int**

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

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

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 »

C'est énoncé Java est valide, **vrai** ou **faux** ?

```
Integer i = 1;
```

« Quiz »

C'est énoncé Java est valide, **vrai** ou **faux** ?

```
Integer i = 1;
```

- S'il est valide, quelles conclusions en tirez-vous ?

« Quiz »

C'est énoncé Java est valide, **vrai** ou **faux** ?

```
Integer i = 1;
```

- S'il est valide, quelles conclusions en tirez-vous ?
- 1 est une valeur d'un type primitif, **int**, mais **i** est une variable référence de type **Integer**

« Quiz »

C'est énoncé Java est valide, **vrai** ou **faux** ?

```
Integer i = 1;
```

- S'il est valide, 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**

« Quiz »

C'est énoncé Java est valide, **vrai** ou **faux** ?

```
Integer i = 1;
```

- S'il est valide, 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 ou 7, cet énoncé est valide !

« Quiz »

C'est énoncé Java est valide, **vrai** ou **faux** ?

```
Integer i = 1;
```

- S'il est valide, 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 ou 7, cet énoncé est valide !

« Quiz »

C'est énoncé Java est valide, **vrai** ou **faux** ?

```
Integer i = 1;
```

- S'il est valide, 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 ou 7, cet énoncé est valide ! Pourquoi ?

Auto-boxing

Java 5, 6 et 7 transforme **automagiquement** l'énoncé

```
Integer i = 1;
```

en celui-ci

```
Integer i = new Integer( 1 );
```

Auto-boxing

Java 5, 6 et 7 transforme **automagiquement** l'énoncé

```
Integer i = 1;
```

en celui-ci

```
Integer i = new Integer( 1 );
```

C'est ce qu'on appelle **auto-boxing**

Auto-unboxing

De même, l'énoncé **i = i + 5**

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

se transformé comme ceci

```
i = new Integer( 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.

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**,

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 ?

```
long s1 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s1 = s1 + (long) 1;
}
```

```
Long s2 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s2 = s2 + (long) 1;
}
```

Dois-je m'en préoccuper ?

```
long s1 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s1 = s1 + (long) 1;
}
```

49 milli-secondes

```
Long s2 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s2 = s2 + (long) 1;
}
```

340 milli-secondes

Dois-je m'en préoccuper ?

```
long s1 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s1 = s1 + (long) 1;
}
```

49 milli-secondes

- Pourquoi ?

```
Long s2 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s2 = s2 + (long) 1;
}
```

340 milli-secondes

Dois-je m'en préoccuper ?

```
long s1 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s1 = s1 + (long) 1;
}
```

49 milli-secondes

```
Long s2 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s2 = s2 + (long) 1;
}
```

340 milli-secondes

- Pourquoi ?

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

```
s2 = s2 + (long) 1;
```

est réécrit (automatiquement) comme ceci

```
s2 = new Long( s2.longValue() + (long) 1 );
```

Astuce de programmation : tests de performance

```
long start, stop, elapsed;

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

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

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

elapsed = stop - start;
```

Astuce de programmation : tests de performance

```
long start, stop, elapsed;

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

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

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

elapsed = stop - start;
```

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

Billion-dollar mistake (Null reference)

≪ I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. **But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.** This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. ≫

Tony Hoare

Opérateurs de comparaison – types primitifs de données

Les opérateurs de comparaison comparent les valeurs !

```
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 ?

Opérateurs de comparaison – types primitifs de données

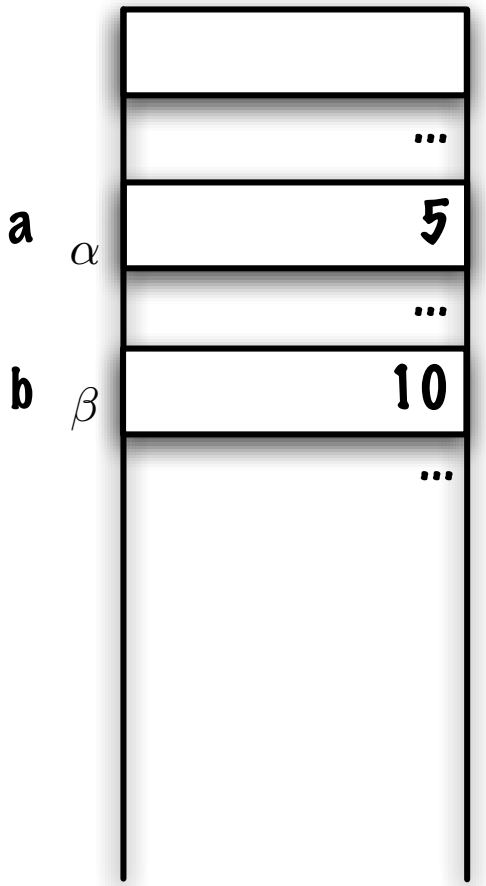
Les opérateurs de comparaison comparent les valeurs !

```
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 « $a < b$ »

```
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 ?

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

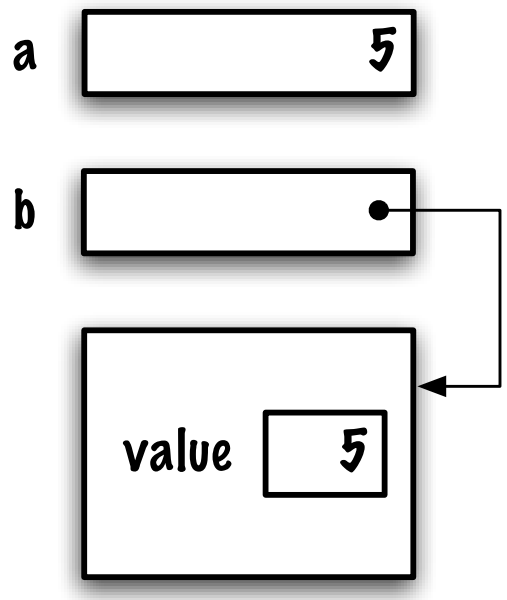
Opérateurs de comparaison : types primitifs et référence

Quel est le résultat ?

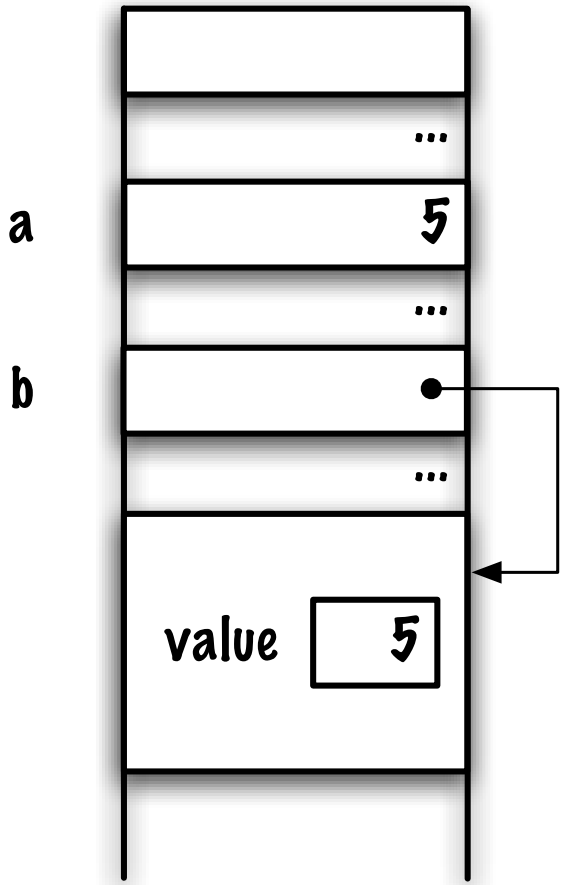
```
int a = 5;
Integer b = new Integer( 5 );
if ( a < b ) {
    System.out.println( "a < b" );
} else if (a == b) {
    System.out.println( "a == b" );
} else {
    System.out.println( "a > b" );
}
```

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

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

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

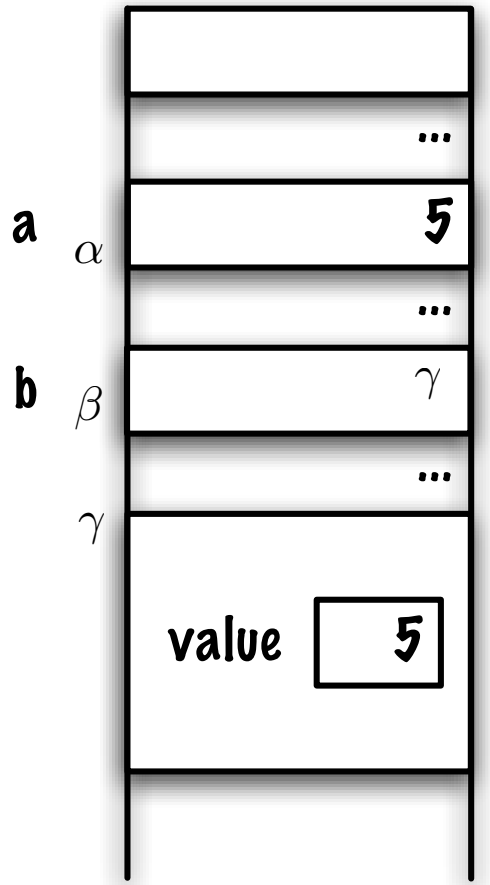



```
References.java:7: operator < cannot be applied to int,java.lang.Integer
        if (a < b)
              ^
References.java:9: operator == cannot be applied to int,java.lang.Integer
        else if (a == b)
                   ^
2 errors
```




```java
int a = 5;
Integer b = new Integer( 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,java.lang.Integer
        if (a < b)
              ^
References.java:9: operator == cannot be applied to int,java.lang.Integer
        else if (a == b)
                   ^
2 errors
```




```java
int a = 5;
Integer b = new Integer( 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,java.lang.Integer
        if (a < b)
              ^
References.java:9: operator == cannot be applied to int,java.lang.Integer
        else if (a == b)
                    ^
2 errors
```

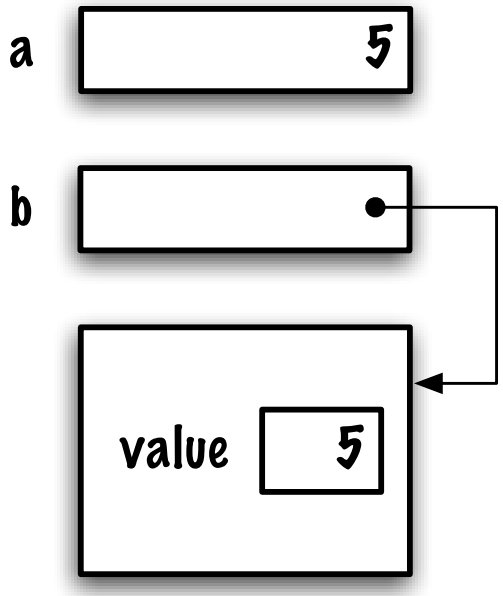
Remarques

- Ces messages d'erreurs sont produits par les compilateurs pré-1.5
- Pour 1.5 et +, l'autoboxing masquera 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;
    }
}
```

```
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

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" );
}
```

$\Rightarrow$ 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" );
}
```

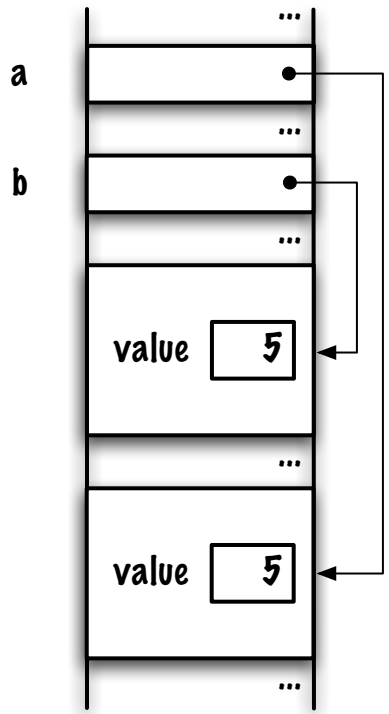
Opérateurs de comparaison et types références

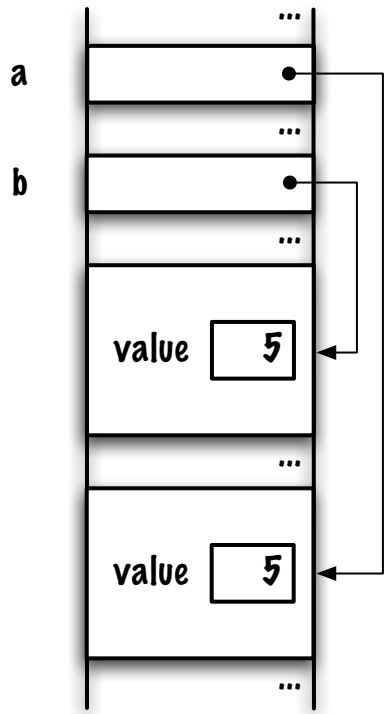
Que se passera-t-il et pourquoi ?

```
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" );
}
```

$\Rightarrow$ 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" );
}
```

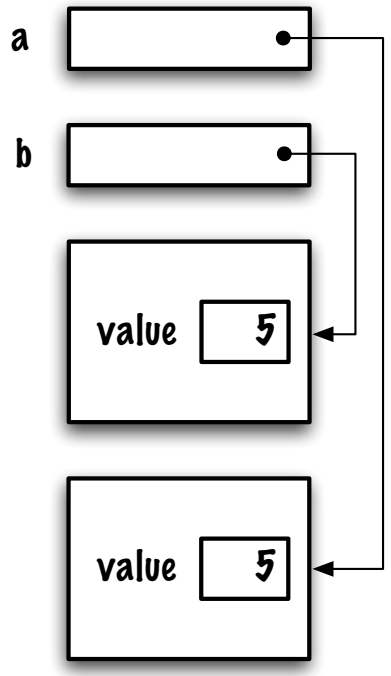





```
MyInteger a = new MyInteger( 5 );
MyInteger b = new MyInteger( 5 );

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

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 ) {
  returns this.value == other.value;
}
```

$\Rightarrow$ 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" );
}
```

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" );
}
```

$\Rightarrow$ Affiche « **a == b** », pourquoi ?

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" );
}
```

$\Rightarrow$ 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" );
}
```

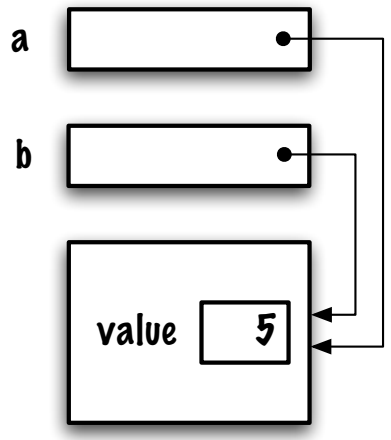
Opérateurs de comparaison et types références

Quel sera le résultat ?

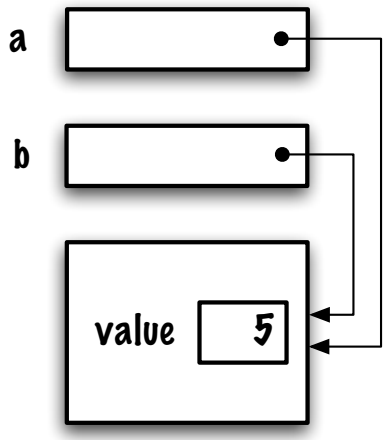
```
MyInteger a = new MyInteger( 5 );
MyInteger b = a;

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

$\Rightarrow$ 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");
}
```



```
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" );
}
```

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**[‡]

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

vs

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

‡. À suivre...

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**.

Définition : arité

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

```
MyInteger() {
  this.value = 0;
}
MyInteger( int v ) {
  this.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

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.

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 :

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

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 d'activation
  (qui contient les paramètres formels et variables locales)

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 d'activation
  (qui contient les paramètres formels et variables locales)
- **les valeurs des paramètres actuels sont affectées aux paramètres formels**

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 d'activation
  (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

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 d'activation
  (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

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 d'activation
  (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 d'activation est détruit)

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 d'activation
  (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 d'activation est détruit)
- l'exécution de la méthode appelante se poursuit avec l'instruction qui suit l'appel de méthode

```
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 );
  }
}
```

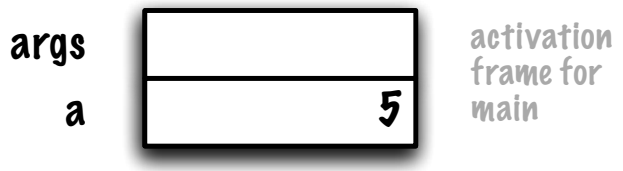
Quel sera le résultat ?

```
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 ?

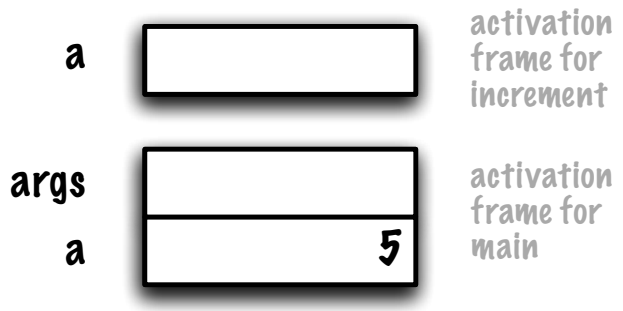
```
before: 5
after: 5
```

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



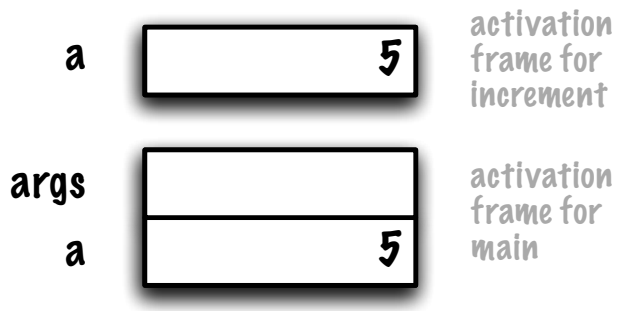
Chaque **appel de méthode** possède sont propre **bloc d'activation**, afin de sauvegarder les paramètres et variables locales à cet appel (ici, **args** et **a**)

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



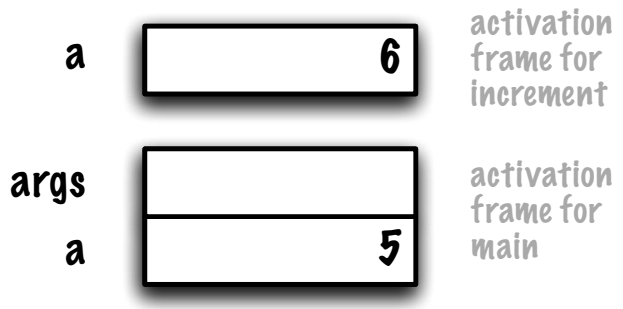
Lors de l'appel à la méthode **increment** un nouveau bloc d'activation est créé

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



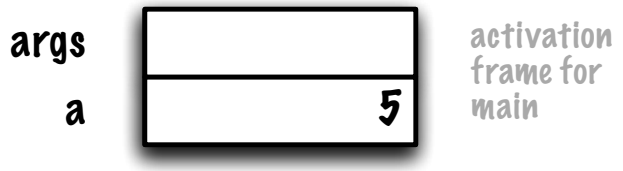
La valeur de chaque **paramètre actuel** est copié à l'emplacement du **paramètre formel** correspondant

```
   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 **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 );
> }
```



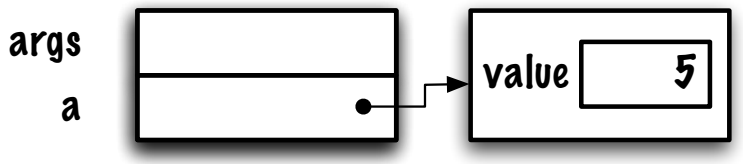
Le contrôle retourne à la méthode **main**, le bloc d'activation pour la méthode **main** est détruit

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);
    }
}
```

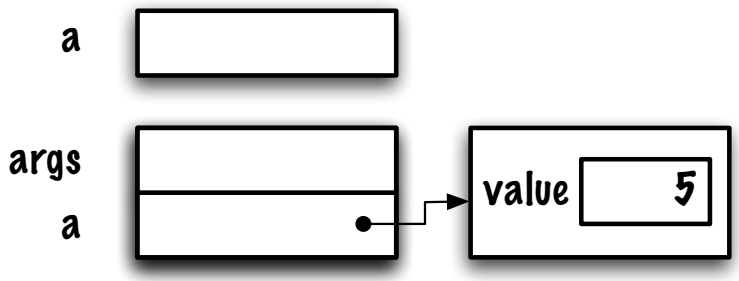
Quel sera le résultat ?

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



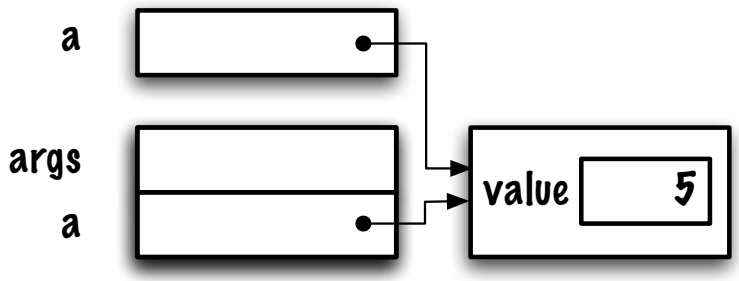
La variable locale **a** de la méthode **main** est une référence vers un objet de la classe **MyInteger**

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



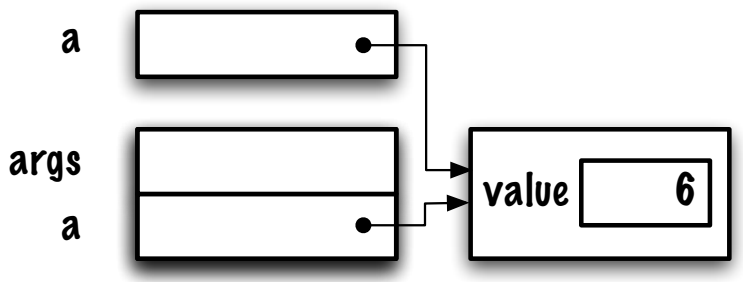
Appel à **increment**, création d'un bloc d'activation

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



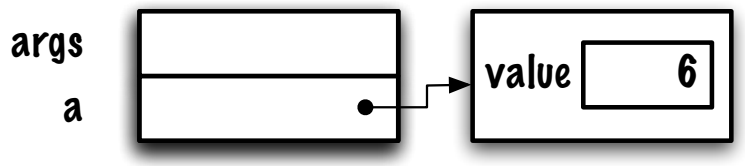
Copier la **valeur** du **paramètre actuel** à l'emplacement du **paramètre formel**

```
  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**

Définition : portée

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

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

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

```
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 ?

```
public class Test {
    public static void main( String[] args ) {
        System.out.println( sum );
        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 ?

```
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;
```

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

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**

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

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

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 immunisés contre les fuites de mémoire, à suivre...

Résumé

- Le typage fort permet la détection de certaines erreurs tôt

Résumé

- Le typage fort permet la détection de certaines erreurs tôt
- Les opérateurs de comparaison compare la valeur des expressions

Résumé

- Le typage fort permet la détection de certaines erreurs tôt
- Les opérateurs de comparaison compare la valeur des expressions
- Lorsqu'on compare des références, on vérifie **que références désignent le même objet ou non**

Résumé

- Le typage fort permet la détection de certaines erreurs tôt
- Les opérateurs de comparaison compare la valeur des expressions
- Lorsqu'on compare des références, on vérifie **que références désignent le même objet ou non**
- On doit utiliser la méthode **equals** afin de comparer le **contenu des objets**

Résumé

- Le typage fort permet la détection de certaines erreurs tôt
- Les opérateurs de comparaison compare la valeur des expressions
- Lorsqu'on compare des références, on vérifie **que références désignent le même objet ou non**
- On doit utiliser la méthode **equals** afin de comparer le **contenu des objets**
- En Java, le passage des paramètres est par valeur (appel-par-valeur)

Résumé

- Le typage fort permet la détection de certaines erreurs tôt
- Les opérateurs de comparaison compare la valeur des expressions
- Lorsqu'on compare des références, on vérifie **que références désignent le même objet ou non**
- On doit utiliser la méthode **equals** afin de comparer le **contenu des objets**
- En Java, le passage des paramètres est par valeur (appel-par-valeur)
- La portée des variables et paramètres est statique en Java

Prochain cours

- Introduction à la programmation orientée objet
  - Rôle des abstractions
  - Activités du développement de logiciels
  - UML – Unified Modeling Language
  - Exemple : Counter

References I


E. B. Koffman and Wolfgang P. A. T.
Data Structures : Abstraction and Design Using Java.
John Wiley & Sons, 2e edition, 2010.


P. Sestoft.
Java Precisely.
The MIT Press, second edition edition, August 2005.


James Gosling, Bill Joy, Guy Steele, and Gilad Bracha.
Java Language Specification.
Addison Wesley, 3rd edition, 2005.

Please don't print these lecture notes unless you really need to !