ITI 1121. Introduction to Computing II

Data types: operator and method calls

by

Marcel Turcotte

Version January 19, 2020

Preamble

Preamble

Overview

Overview

Data types: operator and method calls

We examine the advantages of strongly typed programming languages. We compare primitive types and reference types at the level of comparison operators and method calls.

General objective :

- This week you will be able to contrast primitive types and reference types at the level of comparison operators and method calls.

Preamble

Learning objectives

Objectifs d'apprentissage

- **Compare** the evaluation of primitive and reference type expressions.
- **Enumerate** the steps in a method call.
- **Compare** method calls depending on the case where the parameters are of primitive type and the case where the parameters are of reference type.

Readings:

- Pages 545-551, 571-572 of E. Koffman & P. Wolfgang.

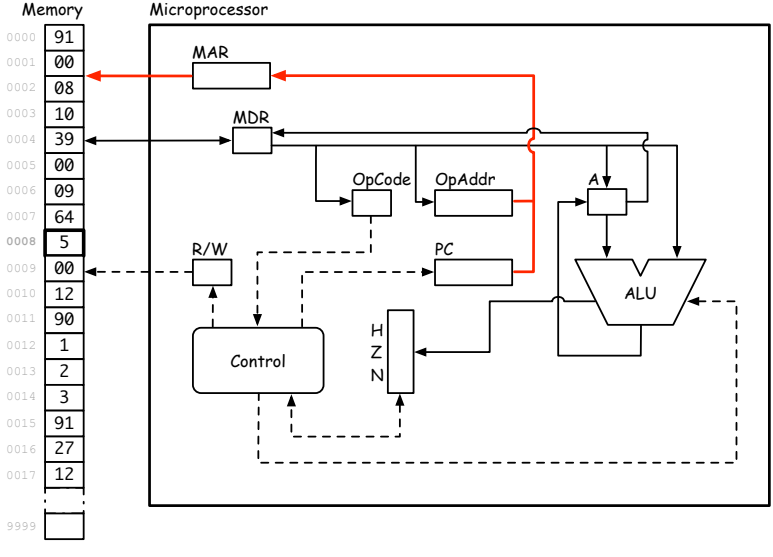
Preamble

Plan

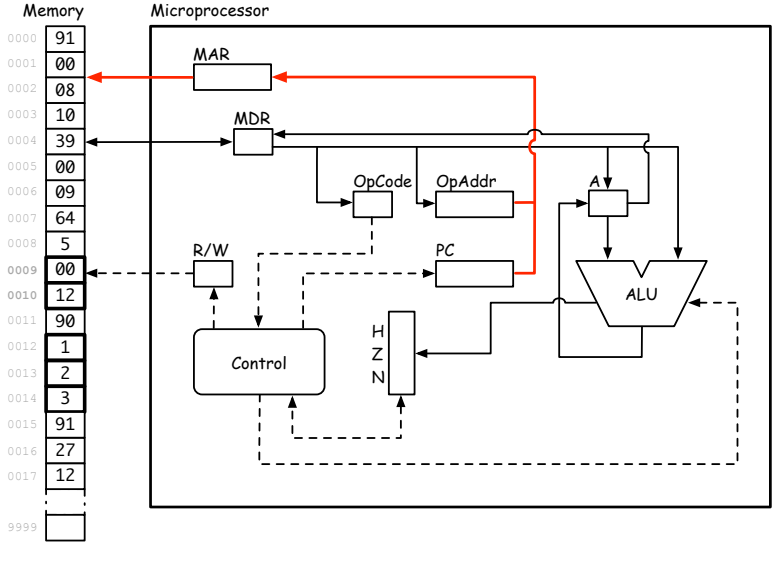
Plan

Reminder : Primitive vs reference and the TC-1101

```
int pos;
pos = 5;

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



- The variable **pos** is of type **int**, a primitive type, if **pos** designates the address **00 08**, then the value **5** is saved at the address **00 08**.



- The variable **xs** is of type reference to an array of integers, if **xs** is the address **00 09**, then the value of the cells **00 09** and **00 10**, is the address where the array was saved in memory, **00 12**. At address **00 12** is the array, with its three values **1**, **2**, and **3**.

Memory diagrams

Memory diagram

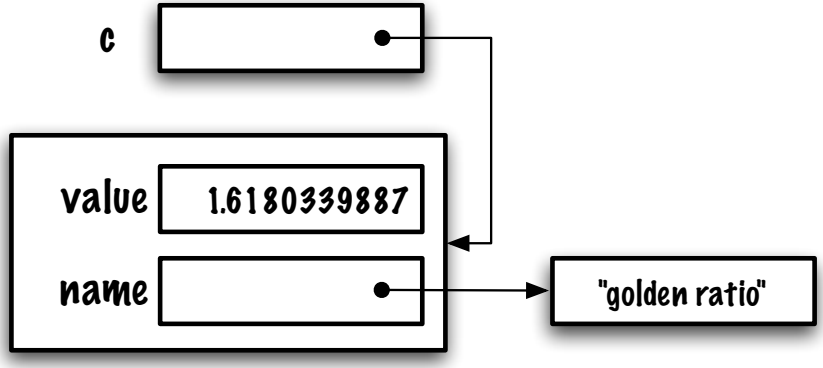
Given the following class declaration:

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

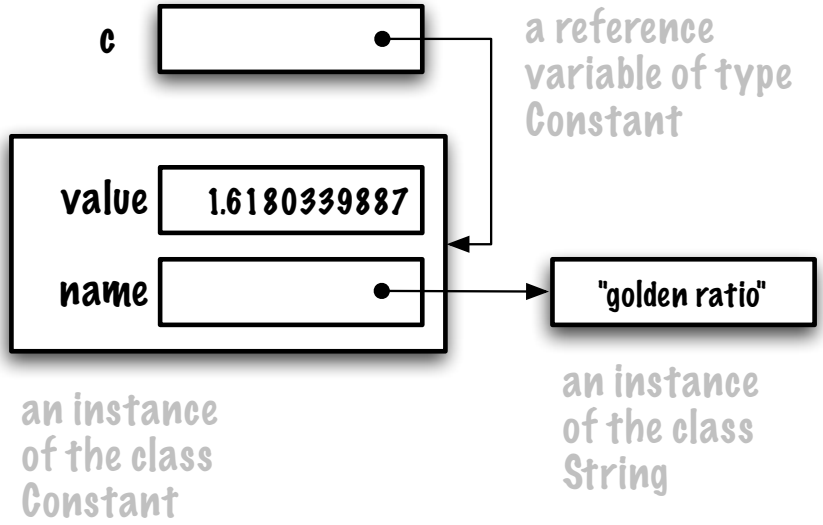
Draw the memory diagram that corresponds to these statements:

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

Memory diagram



Memory diagram



Classe Integer

For the following few examples, we will use a class named **Integer**:

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

Utilisation:

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

Dot notation is used to access the instance variables of an object.

Class Integer

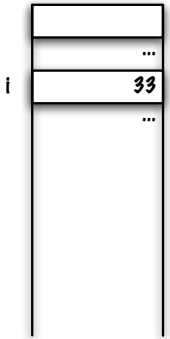
Adding a constructor:

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

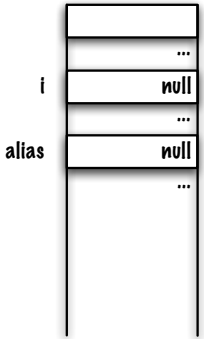
Utilisation:

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

Primitive and reference types



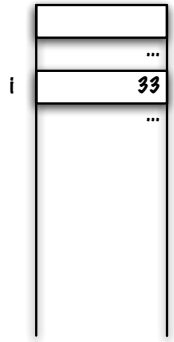

```
int i;
i = 33;
```



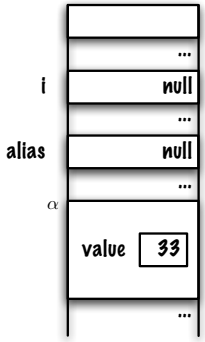

```
Integer i, alias;
i = new Integer(33);
alias = i;
```

On the right, during **compilation**, a portion of memory is reserved for the reference variables **i** and **alias**.

Primitive and reference types



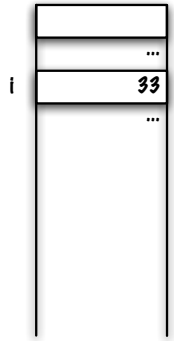

```
int i;
i = 33;
```



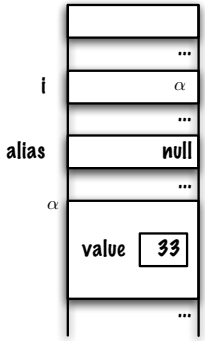

```
Integer i, alias;
i = new Integer(33);
alias = i;
```

Creating an object during the **execution** "**new Integer(33)**".

Primitive and reference types



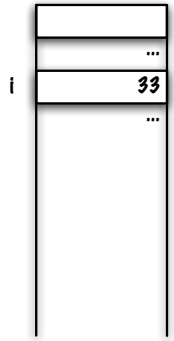

```
int i = 33;
```



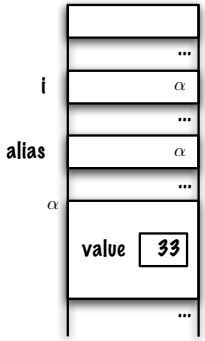

```
Integer i, alias;
i = new Integer(33);
alias = i;
```

Save the **reference** of this object in the reference variable **i**.

Primitive and reference types



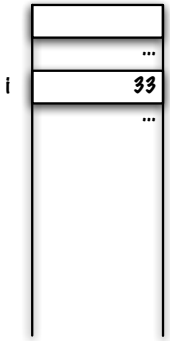

```
int i;
i = 33;
```



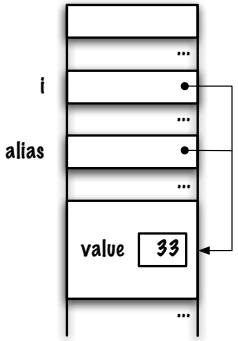

```
Integer i, alias;
i = new Integer(33);
alias = i;
```

Copy the **value** of the reference variable **i** into the variable **alias**.

Primitive and reference types




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

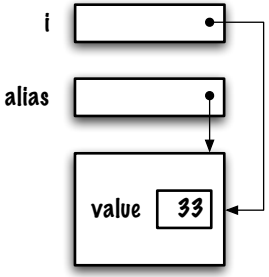



```
Integer i, alias;
i = new Integer(33);
alias = i;
```

i and **alias** refer to the same object!

Primitive and reference types


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




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

Diagramme de mémoire.

« Wrappers »

“Wrapper” classes

- For each **primitive type** there is an associated **wrapper class**.
- **Integer** is the wrapper class for the type **int**.
- A **wrapper object** “stores” a value of a primitive type in an object.
- We’ll use them with stacks, queues, lists and trees.
- Wrapper classes also have several methods for data conversion, e.g. **Integer.parseInt("33")**.

“Quiz”

Are those Java statements valid, **true** or **false**?

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

- **If they’re valid**, what conclusions do you draw from them?
- 1 is a value of a primitive type, **int**, but **i** is a reference variable of type **Integer**.
- For Java 1.2 or 1.4, this statement will produce a compilation **compile time error**.
- However, for Java 5, or newer version, the statement is valid! Why?

Auto-boxing

Java 5, or newer, transform **automagically** the statement

```
Integer i = 1;
```

into this one

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

Where **valueOf** "[r]eturns an **Integer** instance representing the specified int value"*. We call this **auto-boxing**.

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

Auto-unboxing

Likewise, the statement **i = i + 5**:

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

is transformed into this one:

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

where the value of the wrapper object designated by **i** is extracted, **unboxed**, by the call **i.intValue()**.

Boxing/unboxing

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

The eight primitive types have their associated **wrapper**. The automatic conversion from a primitive type to a reference type is called **boxing**,and the conversion from a reference type to a primitive type is called **unboxing**.

Should I be concerned?

```
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?

On the right side, **s2** is of type **Integer**, so the statement

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

is rewritten (automatically) like this

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

Programming Tip: performance testing

```
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;
```

where **System.currentTimeMillis()** returns the number of seconds that have elapsed since midnight, January 1, 1970 UTC (Coordinated Universal Time).
System.nanoTime() also exists!

Operators

Operators

Comparison operators

Comparison operators: primitive data types

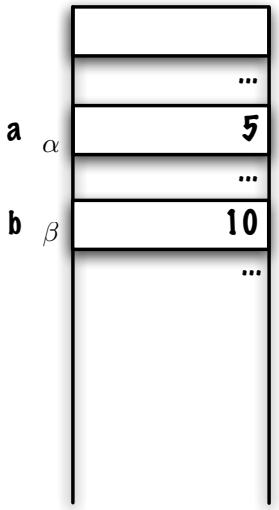
Comparison operators compare the values!

```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");
}
```

What result will be printed out?

$\Rightarrow$ 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");
}
```

Comparison operators: primitive and reference types

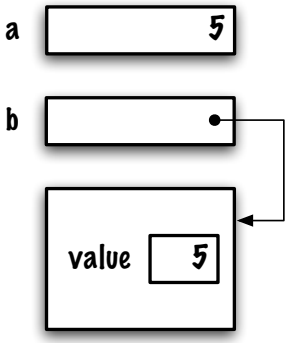
What's the result?

```
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
```

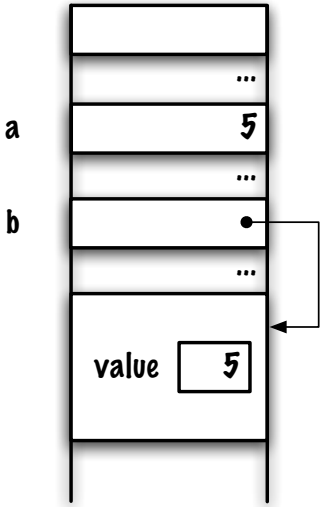
```java
int a = 5;
Integer b = Integer.valueOf(5);

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
```



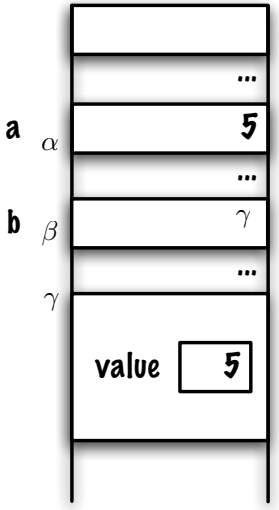
```java
int a = 5;
Integer b = Integer.valueOf(5);

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
```

Remarks

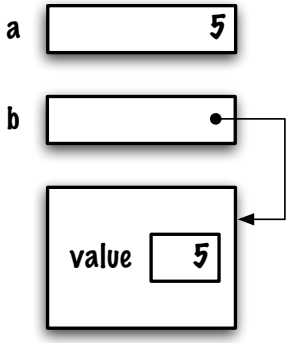
- These error messages are produced by pre-1.5 compilers.
- For 1.5 and up, autoboxing will (possibly) hide the “problem”.
- To get the same behavior for both environments, let’s use our own wrapper class, **MyInteger**.

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");
}
```

- Correct these statements!



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** »

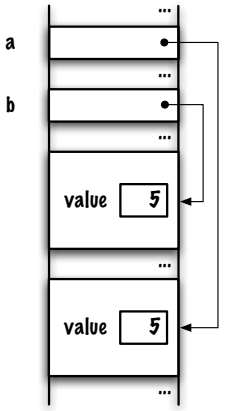
Comparison operators and reference types

What will happen and why?

```
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");
}
```

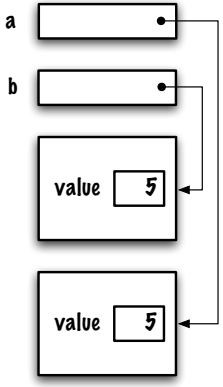
⇒ Prints « **a does not equal b** »

```
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");
}
```



⇒ Prints « **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");
}
```



⇒ Prints « **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");
}
```

where the **equals** method would have been defined as follows

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

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

What's the result?

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

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

⇒ Displays " **a == b** ", why? because the references **a** and **b** designate the same object, the same instance, in other words, the two memory addresses are identical; we say that **a** and **b** are aliases.

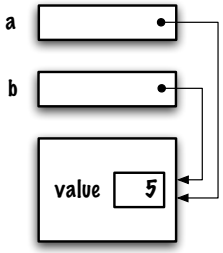
Comparison operators and reference types

What's the result?

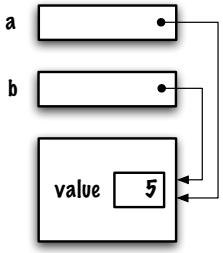
```
MyInteger a = new MyInteger(5);
MyInteger b = a;

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

⇒ Prints « **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");
}
```



Comparison operators and reference types

What's the result?

```
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
```

Remarks

- To compare the contents of the designated objects, “content equivalence” or “logical equivalence”, we use **equals**[†]
- To know if two reference variables **designate the same object**, we use the comparison operators **'=='** and **'!='**.

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

vs

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

[†]To be continued...

Exercises

Compare objects two by two using either **equals** or ==, you could be surprised.

```
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 + "";
```

In particular, try **a == b** and **e == f**.

Method call

Method call

Definitions

Definition: arity

The **arity** of a method is the number of parameters; a method has none, one or more parameters.

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

Definition: formal parameter

A **formal parameter** is a variable that is part of the signature of the method; it can be seen as a local variable in the body of the method.

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

⇒ **a** and **b** are the formal parameters of **sum**.

Definition: actual parameter

The **actual parameter** is the value that is used during the method call and provides the initial value to the **formal parameter**.

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

midTerm and **finalExam** are the actual parameters of the call to the method **sum**, during the call the **value** of the actual parameter is copied to the location of the formal parameter.

Concept: call-by-value

In Java, during a method call, the **value** of the actual parameter is **copied** to the location of the formal parameter.
When a method is called:

- the execution of the calling method is interrupted
- a block of working memory is created [‡] (which contains the formal parameters and local variables)
- **The values of the actual parameters are assigned to the formal parameters.**
- the body of the method is executed
- the return value is saved
- (the working memory block is destroyed)
- the execution of the calling method continues with the instruction that follows the method call

[‡] activation frame

Method call

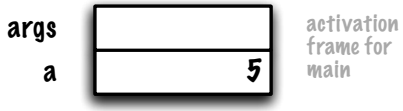
Primitive types

```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);
  }
}
```

What's the result?

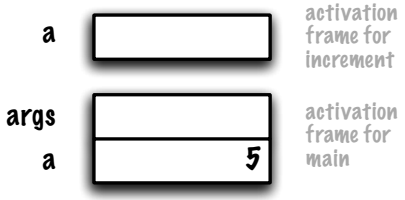
```
before: 5
after: 5
```

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



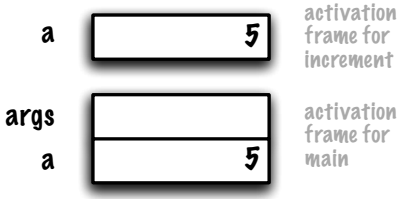
Each **method call** has its own working memory (**activation block**), which serves to save the parameters and local variables for this call (here, **args** and **a**).

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



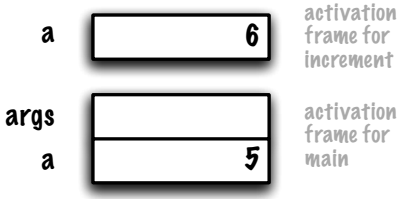
When calling the method **increment** a new block of working memory is created.

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



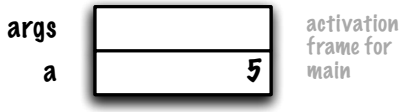
The value of each **actual parameter** is copied to the location of the corresponding **formal parameter**.

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



The execution of the statement $\mathbf{a = a + 1}$ changes the value of the formal parameter **a**, a memory location distinct from that of the local variable **a** of the method **main**.

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



The control returns to the method **main**, the working memory for the method **increment** is destroyed.

Method call

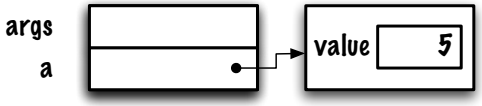
Reference variables

References and method calls

```
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);
    }
}
```

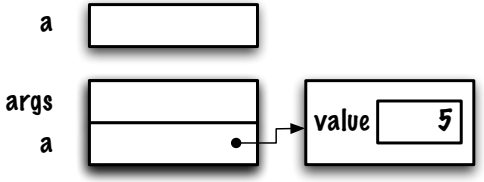
What's the result?

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



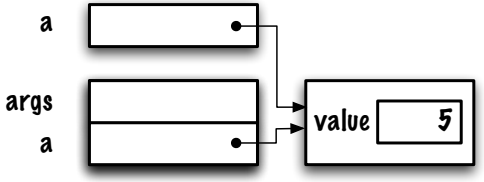
The local variable **a** of the method **main** is a reference to an object of the class **MyInteger**.

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



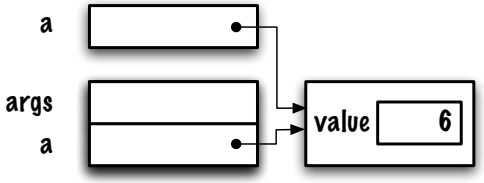
Calling **increment**, creating an activation block.

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



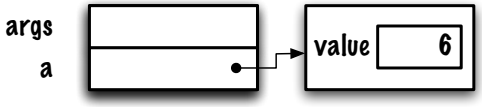
Copy the **value** from the **actual parameter** to the location of the **formal parameter**.

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



Executing **a.value++**

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



Return the control to the method **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]);
        }
}
```

Scope

Scope

Definitions

Definition: scope

*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.

Definition: scope of a local variable in 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: scope of a parameter in Java

*The scope of a **formal parameter** of a method (§8.4.1) or constructor (§8.8.1) is the entire body of the method or constructor.*

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

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

Scope

Examples

```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();
    }
}
```

Is this a valid Java program?

```
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;
        }
    }
}
```

Is this a valid Java program?

```
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;
        }
    }
}
```

Is this a valid Java program?

Memory management

What happens to the objects when there's **no reference to them**? Here, what happens to the object holding the value 99?

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

- The JVM will recover the associated memory space!
- This process is called **garbage collection**
- Some programming languages do not automatically handle memory allocations and de-allocations.

Java, however, is not immune to memory leaks, to be continued...

Prologue

Summary

- Comparison operators compare the values of the variables.
  - In particular, if **a** and **b** are reference variables, then **a == b** returns **true** ssi **a** and **b** designates the same object.
- Method calls are **per value** in Java.

Next module

- **Object-oriented programming**

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

School of Electrical Engineering and **Computer Science** (EECS)

University of Ottawa