# Chapter II:

## Class and Object

Mila University Center
2nd Year Computer Science Degree
Subject: Object Oriented Programming
Head of subject : Dr. Sadek Benhammada

# 1. Class declaration syntax

# 1. Class declaration syntax

Header

Body

```
[ Modifiers ] class ClassName [ extends mother_class ] [ implements [interfaces]

{
  //Attributes
    [ Modifiers ] type nameAttribute_1;
    [ Modifiers ] kind nameAttribute_2;
   ...
   //Methods
    [ Modifiers ] ReturnTypeMethodName_1 ( params )
    {
      // method body;
    }
   [ Modifiers ] ReturnType methodName_2 ( params )
     {
    // method body;
     }
   ...
}
```

# 1. Class declaration syntax

- A class consists of two parts: (1) **header** and (2) **body** .

**1.1 . The header:**

- Modifiers class (optional) are: **abstract** , **final** , and visibility ( **private** , **public**
- The keyword **class** followed by the name of the class (required ) **;**
- The keyword **extends** followed by the **name of the superclass** (optional );
- The keyword **implements** followed by the list of interface names ( optional);

**Examples:**

```
public class Form {…}
public class Rectangle extends Shape{…}
```

**1.2. The body:** surrounded by opening and closing braces ( **{** … **}** ), it contains the declarations of **attributes** and **methods:**

# 1. Class declaration syntax

### 1.3. Declaring an attribute (in order):

- **Modifiers** (optional): ***static*** , ***final*** , and visibility ( ***private*** , ***protected*** , ***public*** );
- **Type** : The type is either:
  - a basic type of the language, ( ***boolean*** , ***byte*** , ***short*** , ***int*** , ***long*** , ***float*** , ***double*** , ***char*** , ***void*** ),
  - or the name of another class in the program.
- **Name** : name of the attribute

**Examples:** Attribute Declaration

```
private int x;
public static final PI=3.14;
```

# 1. Class declaration syntax

**1.4. Declaration of a method:** The declaration of a method is composed of the **signature** and the **boy :**

- **The signature** :
  - **Modifiers** (optional): ***abstract*** , ***static*** , ***final*** , and visibility ( ***private*** , ***protected*** , ***public*** );
  - **The return type** of the method;
  - **Method name;**
  - And **the method parameters;**
- **The body:** a series of instructions placed between { }.

**Example:** declaring a method

```
public double sum(double x, double y) {
 double s= x+y ;
return s;
}
```

# 1. Class declaration syntax

- **1.5. Basic types**

| Types | Size | values | Example |
|---|---|---|---|
| **byte** | 1 byte | Integers between -128 and +127 | byte temperature ;<br>temperature = 64; |
| **shorts** | 2 bytes | Integers between -32768 and +32767 | short speedMax ;<br>speedMax = 32000; |
| **int** | 4 bytes | Integers between -2147483648 and 2147483647 | int temperatureSun ;<br>temperatureSun = 15600000; |
| **long** | 8 bytes | Integers between<br>- 9223372036854775808 and 9223372036854775807 | long yearLight ;<br>lightyear =9460700000000000; |
| **float** | 4 bytes | Floating point numbers between 1.401e-045 and 3.40282e+038 | float pi;<br>pi = 3.141592653f ; |
| **double** | 8 bytes | Floating point numbers between 2.22507e-308 and 1.79769e+308 | double division ;<br>division = 0.3333333333334 ; |
| **char** | 2 bytes | character (65000 characters possible) | char character ;<br>character = 'A' |
| **boolean** | 1 bit | logical value: true or false | boolean question;<br>question = true |

# 1. Class declaration syntax

## 1.6. Character strings

- Strings in Java do not correspond to a data type but to a **`String class`** .
- A string can therefore be declared as follows:

```
String sentence = " Hellow world " ;
```

- `sentence` **is not a variable** but an **object** of the **class `String`** .
- Java supports the + operator as a string concatenation operator .
- The + operator allows to concatenate several character strings.

**Examples:** Declaration and concatenation of character strings

```
String s1=" Hello";
String s2="World";
String s3=s1+s2;// s3== Hellow world
```

# 1. Class declaration syntax

- **1.7.Type Conversion in Java**
- **Java is Strongly Typed**
  - Java enforces **strict type-checking** at **compile-time**
  - Implicit type conversions that may **lead to data loss** are **not allowed**.

**Example:**

- This is **valid in C** but **invalid in Java**:

```
double a = 5.5;
int y = a; // Allowed in C
```

- In Java, an **explicit cast** is required:

```
double a = 5.5;
int y = (int) a; // Explicit type conversion
```

# 1. Class declaration syntax

- **1.7.Type casting in Java**
- In Java, *type casting* refers to the process of converting a value from one data type to another ( byte , short , int , long , float , double , char )
- The cast can be implicit or explicit

**1.7.1 Implicit Type casting**

- Implicit type conversion occurs automatically when a value of a smaller data type is assigned to a larger data type :

byte (1 bytes) → short (2 bytes) → int (4 bytes) → long (8 bytes) → float (4 bytes) → double (8 bytes).

**Example of implicit casting**

```
int i =5;
double d = i; // Implicit conversion from int to double
System.out.println(d); // Outputs: 5.0
```

# 1. Class declaration syntax

## 1.7.Type casting in Java

### 1.7.2 Explicit Type Casting

- Explicit casting is required when converting a value from a larger data type to a smaller one.
- This prevents unintended data loss and improves code reliability.
- **Example:**

```
double d = 10.75;
int i = (int) d; // Explicit conversion from double to int
System.out.println(i); // Outputs: 10
```

- The fractional part of the double value is truncated during conversion to int, resulting in potential loss of information.

# 1. Class declaration syntax

- **Example :** Declaration of a class **Point**

```
audience class Point {
// attributes
    private double x ; // Abscissa
    private double y ; //Ordinate
// methods
public String toString (){
    return "Point(" + x + "," + y + ")" ;
}
}
```

# 2. Java Naming Conventions

# 2. Java Naming Conventions

1. **Use meaningful names** for classes, attributes, methods and variables . The name should be sufficient to understand what a method does, for example, without seeing the code's details.
2. **Class names** start with a **capital letter** ,

   **Examples:**

   ```
   public class Rectangle {…}
   public class Person {…}
   ```

3. The names of **attributes** , **methods** and **variables** start **with lowercase** .

   **Examples:**

   ```
   private double length; //attribute
   public double surface() {…} //method
   ```

# 2. Notation conventions

1. **Use meaningful names** for classes, attributes, methods and variables . The name should be sufficient to understand what a method does, for example, without seeing the code's details.
2. **Class names** start with a **capital letter** ,

   **Examples:**

   ```
   public class Rectangle {…}
   public class Person {…}
   ```

3. The names of **attributes** , **methods** and **variables** start **with lowercase** .

   **Examples:**

   ```
   private double length; //attribute
   public double surface() {…} //method
   ```

# 2. Notation conventions

4. When a name is made up of several words joined together, each successive name begins with a capital letter.

   **Examples :**

```
public class BankAccount {…} //Class
public int numberWheels ; //attribute
public double calculateSurface (){…}; // method
```

5. The name of a **constant** is in **UPPERCASE** . When the name of a constant consists of several words with the words separated by the underscore character

   **Examples :**

```
public static final double PI =3.14;
public static final int MAX_NUMBER =100;
```

6. Typically , the first word of a method name is a verb.

   **Example** :

```
public double calculateSurface ( )
```

# 3. Declaration and creation of an object

# 3. Declaration and creation of an object

## 3.1. Declaring an object

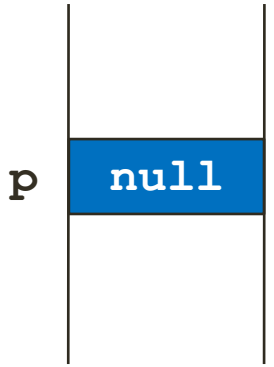
- The declaration of an object is of the form:

```
ClassName objectName ;
```

**Example:** Declaring an object of the Point class

```
Point p;
```

- The declaration of the object ( `Point p` ) reserves a memory location for a ***reference*** on an object of type Point.




- At this point, the value of the variable `p` is ***`null`***

# 3. Declaration and creation of an object

### 3.2. Creating an object

- For that **p** actually references an object, you must call a **constructor** .
- The constructor is the method used to create an object ( *allocate memory space for the object* ) of a given class and possibly *initialize* its attributes.
- The constructor **is named after the class** and **does not mention a return type.**

***Example:*** Constructor of the **Point class**

```
public class Point {
    private double x ;
    private double y ;
```

```
//Constructor
 public Point (double a,double b)
 {
    x=a;
    y=b;
  }
}
```

# 3. Declaration and creation of an object

## 3.2. Creating an object

- To create an object, a constructor is invoked using the **new operator** , which performs the memory reservation and returns the address of the allocated area.

***Example***

```
Point p;//Declaration of the object p
p=new Point(5,3); // Create the p object
```

- It is possible to combine the declaration and creation of an object.

***Example***

```
Point p = new Point(5,3);
```

# 3. Declaration and creation of an object

## 3.2. Creating an object

- It is possible to declare several constructors for the same class (**overload the constructor** ).

***Example:***

We can declare another constructor for the *Point class* , to create objects whose *x* and y attribute values are equal.

```
public Point(double a)
{
 x=a;
 y=a;
}
```

# 3. Declaration and creation of an object

### 3.2. Creating an object

- **The Default Constructor:** If no constructor is written for a given class, it is possible to use the default constructor which simply allocates a memory location ( **it does not initialize the attributes** ).

- For a `Point` class , the default constructor is as follows:

```
public Point (){}
```

***Example***

If the *Point class* has no constructors, we can write:

```
Point p = new Point();
```

**Note:**
If not initialized, a class's attributes are automatically assigned default values:

- **0** for numeric attributes ( **int** , **float** , **double** , etc.),
- **false** for booleans, and
- **null** for objects (Example: **String** type attributes ).

# 3. Declaration and creation of an object

## this

- **The `this` keyword is used to reference the object currently in use in a method.**

- **Example :**

```
// Constructor of the Point class

public Point(double a, double b)
{
      this.x = a;
      this.y = b;
   }
```

- The instruction **`this`** `.x =a;` means that the `x attribute` of the current object ( **`this`** ) is assigned the value `a` .

# 3. Declaration and creation of an object

## this

- When a method of an object references an attribute **x** of this object, writing **this.x** is implicit.
- **this** keyword must be used explicitly when there is **a conflict of identifiers** .

- **Example :**

  We must use the **this** keyword explicitly, when the same identifiers are used for attributes and for constructor parameters .

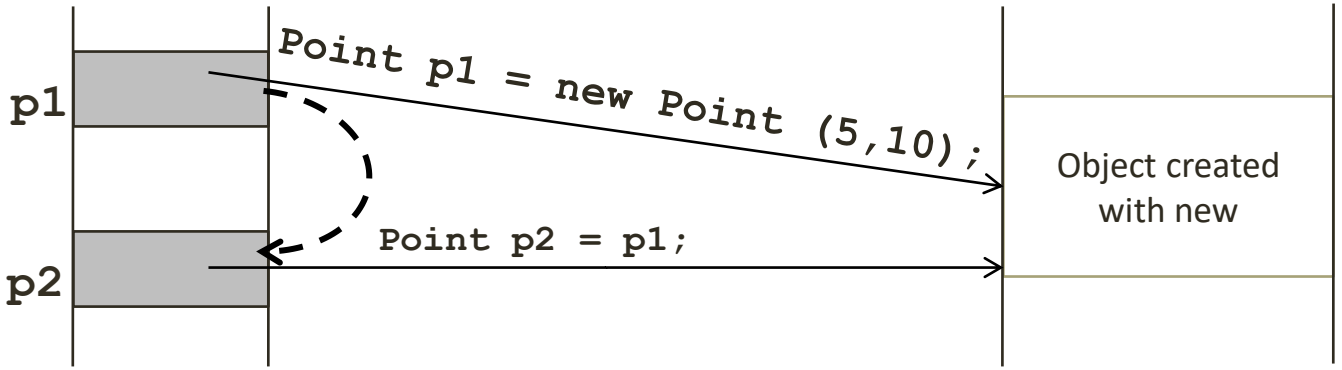
```
// Constructor of the Point class

audience Point(double x, double y)
{
      this .x =x;
      this .y =y;
   }
```

# 3. Declaration and creation of an object

## 3.4. Creating identical objects

- We may need to create two absolutely identical objects.
- Let's look at the following code :

```
Point p1 = new Point();
Point p2 = p1;
```




- **p1** and **p2** contain the same reference ⇒ **p1** and **p2 point to the same object** .
- Changing the values of the attributes of **p1** also changes the values of the attributes of **p2** since, in fact, it is the same object .

# 3. Declaration and creation of an object

## 3.4. Creating identical objects

### Solution : Copy Constructor

- Another solution to create identical objects is to define a copy constructor;
- **Example:** Copy constructor of the `Point class`

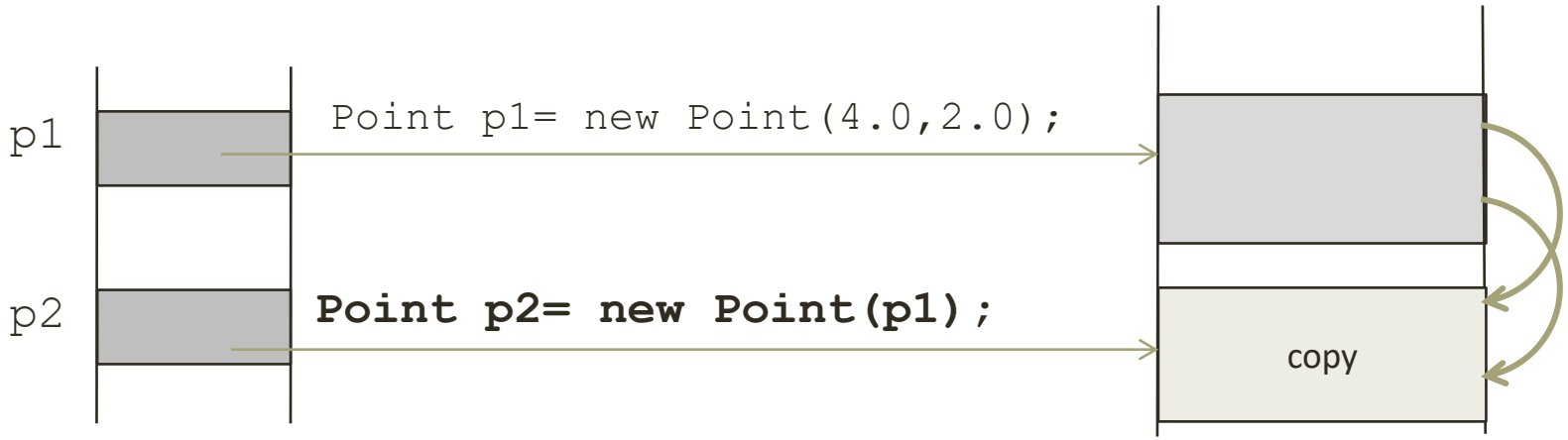
```
audience class Point {
  private double x ;
  private double y ;
     // Constructor
  audience Point( double x, double y)
{
       this . x =x;
       this . y =y;
}
    //COPY Constructor
   public Point (Point p)
   {
      this . x = p. x ;
      this . y = p. y ;
}

 }
```

## 3.4. Creating identical objects

**Solution: Copy Constructor**

- **Example:** Creating an object of the `Point class` using the copy constructor

```
Point p1= new Point(4.0,2.0);
Point p2= new Point(p1);
```



# 3. Declaration and creation of an object

## 3.5. Deleting objects

- Objects are not static elements and their lifetime does not necessarily correspond to the execution time of the program.
- The lifespan of an object goes through three stages :

1. The declaration and creation of the object.
2. Using the object by calling these methods.
3. Object deletion: it is automatic in Java thanks to the memory collector ( `Garbage Collector :` **GC** ).

- GC *is* used to automatically delete objects that are no longer referenced by the program. In C++, it is the programmer who takes care of deleting unnecessary objects.

# 3. Declaration and creation of an object

## 3.5. Deleting objects

- Objects are not static elements and their lifetime does not necessarily correspond to the execution time of the program.
- The lifespan of an object goes through three stages :

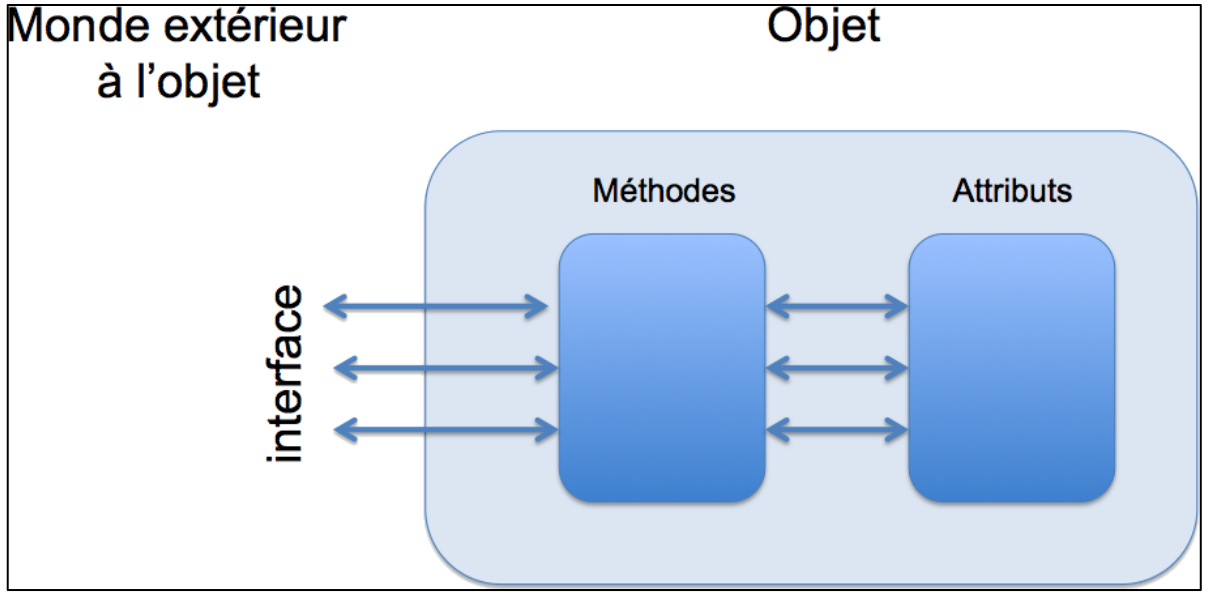
1. The declaration and creation of the object.
2. Using the object by calling these methods.
3. Object deletion: it is automatic in Java thanks to the memory collector ( `Garbage Collector :` **GC** ).

- GC *is* used to automatically delete objects that are no longer referenced by the program. In C++, it is the programmer who takes care of deleting unnecessary objects.

# 4. Encapsulation

# 4. Encapsulation

- Encapsulation is the ability to hide parts of an object's members (attributes and methods), i.e. by preventing direct access to these members from the outside.
- Encapsulation allows you to only show what is necessary for your use of the object .
- The list of methods and attributes that can be used from outside is called the class **'s interface .**



# 4. Encapsulation

## 3.1. Access control to attributes and methods

- To achieve encapsulation, we have a set of modifiers *access* control to *classes* , *methods* and *attributes* .

- For the *methods* and *attributes* Within classes, the Java programmer has 3 levels of access control, which he sets using 3 visibility modifiers .
  - *audience* : public elements are accessible without any restrictions.
  - *protected* : protected elements are only accessible from the class and subclasses that inherit from it.
  - *private* : private elements are only accessible from within the class itself.

- The default visibility, when nothing is specified, is equivalent to ***public*** .

# 4. Encapsulation

## 4.1 . Access control to attributes and methods

- **In general :**
  - **The attributes** of a class are declared ***private.*** ( `private` ) or **protected** , meaning that only objects of the class **or its subclasses can read and modify them**

  - **Methods** are declared ***public*** **, which means** `that` any object can call **them ;**

# 4. Encapsulation

## 4.2. Reading and modification

- To read and modify the attributes of an object, we add *methods* specially designed for this purpose to the class, which we call " *Accessors* " .
- **Read accessors**
  - Getters **are** methods that allow you to **read** the attributes of the object;
  - Getter names usually start with *get followed by the* attribute name **;**

  ***Example*** : `public String getName () {return name}`

- **Modification**
  - **Modification accessors are** methods that allow you to **modify** the **attributes** of the object;
  - The names of modifiers usually begin with *set* followed by the attribute name.

  ***Example*** : `public void setName (String n) {name=n;}`

# 4. Encapsulation

## Example

```
public class Point
{
//Attributes
    private double x;
    private double y ;
//Reader accessors
public double getX (){return x;}
public double getY (){return y ;}
// Modification accessors
    public void setX (double x){ this.x =x;}
public void setY (double y){ this.y =y ;}
toString
public String toString (){
       return "Point("+ x +","+ y +)";
 }
}
```

# 4. Encapsulation

## Example (continued )

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

   Point p= new Point(); /*Creating a Point object using the default
   constructor */

   p.setX (5.0 ); /* Use setX () modifier accessor to initialize x
   attribute */

   p.setY (10.0 ); /* use setY () modifier accessor to initialize y
   attribute */

   System.out.println ( p.getX ()); /* use getX () getter to display
   the value of attribute x */

   System.out.println ( p.getY ()); // use getY () read accessor to
   display the value of the y attribute
  }
}
```

**The result displayed:**

**5.0**
**10.0**

## 4.3. Interest of accessors

- The interest accessors is to make all the rest of the code independent of the representation of the object:

- If we decide to modify an attribute, we only need to modify the code of the accessor itself, that is, a single line of the program, whereas we would have had to modify all the lines where the attribute was used if we had not used an accessor.

# 4. Encapsulation

## 4.3. Interest of accessors (Example)

- In the **`Person`** class , we can declare the **`age attribute`** with **`public`** visibility .
- In this way all objects that make up the system can access and modify the **`age attribute`** objects of the class.
- If we decide to replace the `int` attribute `age` by **`int yearOfBirth`** , wherever the **`age attribute was used`** , the code must be modified .

```
public class Person {
// attributes
 ...
 public int age;
// methods
…
}
```

```
public class Person {
// attributes
 ...
 private int yearOfBirth ;
// methods
…
}
```

```
Public class AClass {
…
Person p=new Person(…);
int x=p.age;
…
p.age=15;

… }
```

```
Public class AClass {
…
Person p=new Person(…);
int x=p.age;
…
p.age=1 5;

… }
```

## 4.3. Interest of accessors (Example )

- If **`getAge ()`** was used and **`setAge ()`** , then just change the accessor code.

```
public class Person {
// attributes
 ...
 private int age;
// method
...
 audience int getAge (){
       return age;
}
 audience void setAge ( int a){
       age =a;
}
}
```

```
Public class AClass {
…
Person p=new Person(…);
int x= p.getAge ();
….
p.setAge (14);

… }
```

```
import java.util.GregorianCalendar ;
public class Person {
     // attributes
     ...
     private int yearOfBirth ;
     // method
      //...
    audience int getAge (){
    GregorianCalendar d = new GregorianCalendar ();
    return d.get ( d. YEAR )- yearofBirth ;
   }

   audience void setAge ( int a){
   GregorianCalendar d = new GregorianCalendar ();
   yearofBirth = d.get ( d. YEAR )-a;
}
}
```

# 5. Packages

# 5. Packages

## 5.1. Definition

- Java classes are grouped into **packages** ;

- Packaging is a means of modularity that allows:

  - Split a large application into packages grouping together classes that cover the same domain;

  - protect attributes and methods;

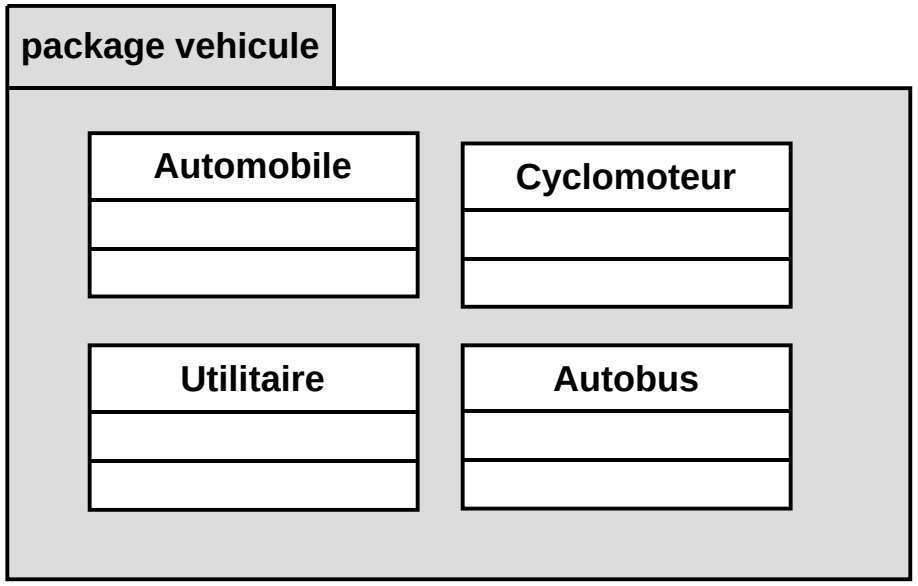
## 5.2. Naming a package

- Each package has a name. By convention, the name of a package begins with a lowercase letter.
- Any class belonging to a package must first declare its membership in that package, using the statement:

```
package packageName ;
```

***Example***

```
vehicle package ;
public class Automobile{...}

vehicle package ;
public class Moped {...}

vehicle package ;
public class Utility {...}

vehicle package ;
public class Bus {...}
```



## 5.3. Class Access Control

- For classes, there are only two levels of visibility:

1. **Public** : The class is visible to classes in its package , and outside the package .
   - The syntax for declaring a public class is to write:

```
public class AClass {...}
```

**Example**

```
public class Automobile {...}
```

2. **No visibility** : The class is only visible to classes in the *package* it is in.
   - The syntax is to write:

```
class AClass {...}
```

**Example**

```
class Point {...}
```

# 5. Packages

## 5.3 . Using a package

- To designate a class that is defined in another package, we have the choice between :

1. Import the class :

```
import PackageName.ClassName ;
```

2. Precede each occurrence of the class name with the name of the package in which it is defined .

**Example**

```
// Declaration of the person class
package owner ;
public class Person{...}
```

# 5. Packages

**Example (continued)**

To designate within the **vehicle package** the **Person class** which belongs to the **owner package** , we have the choice between the following code:

```
package vehicle ;
import owner.Person ;
public class Automobile {
...
Person p;
p=new Person(String firstname , String lastname)
...
}
```

Or the following code:

```
package vehicle ;
public class Automobile {
...
owner. Person p;
p=new owner. Person (String firstname , String lastname)
...
}
```

## 5.3 . Using a package

- For import all classes from a package:

```
import nomPackage .*;
```

**Example**

The following code can be replaced:

```
import vehicle. Automobile
import vehicle . Moped import
vehicle.Utility
import vehicle.Bus
```

By the following code:

```
import vehicle.*
```

# 6. Passing parameters

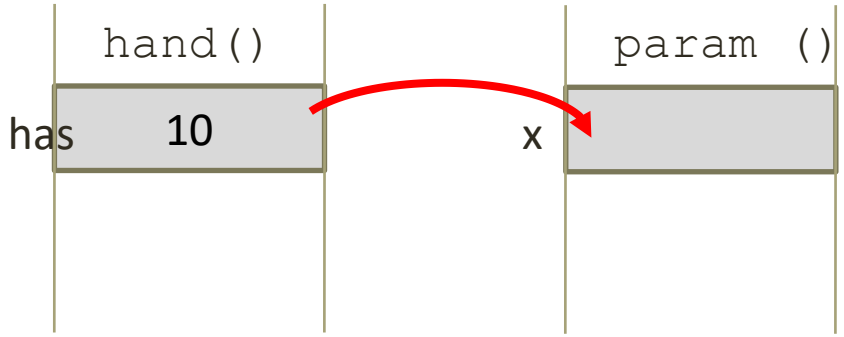
# 6. Passing parameters

- In Java, parameters are always **passed by value** , that is, the value of the actual parameter is copied into the corresponding formal parameter :
  - When each method is called, local memory space is allocated for each formal parameter;
  - The values of the actual parameters are copied before the method is called;
  - The calculation is carried out on the formal parameters ;

# 6. Passing parameters

- **Example**

```
public class Test {

public void param (double x)
{
x=5
   }
public static void main(String arg [])
{
   Test test=new Test();
   double a = 10;
   System.out.println ("Before calling param : a="+a);
   test.param (a);
   System.out.println (" After calling param : a="+a);

}
}
```




Result displayed:
Before calling param : a=10.0
After calling param : a=10.0

# 6. Passing parameters

**Case of the passage of an object**

- When passing an object as a parameter, it is the reference to this object which is passed and copied as a formal parameter.
- So if the object is modified in the method, the changes will be visible from the outside.

# 6. Passing parameters

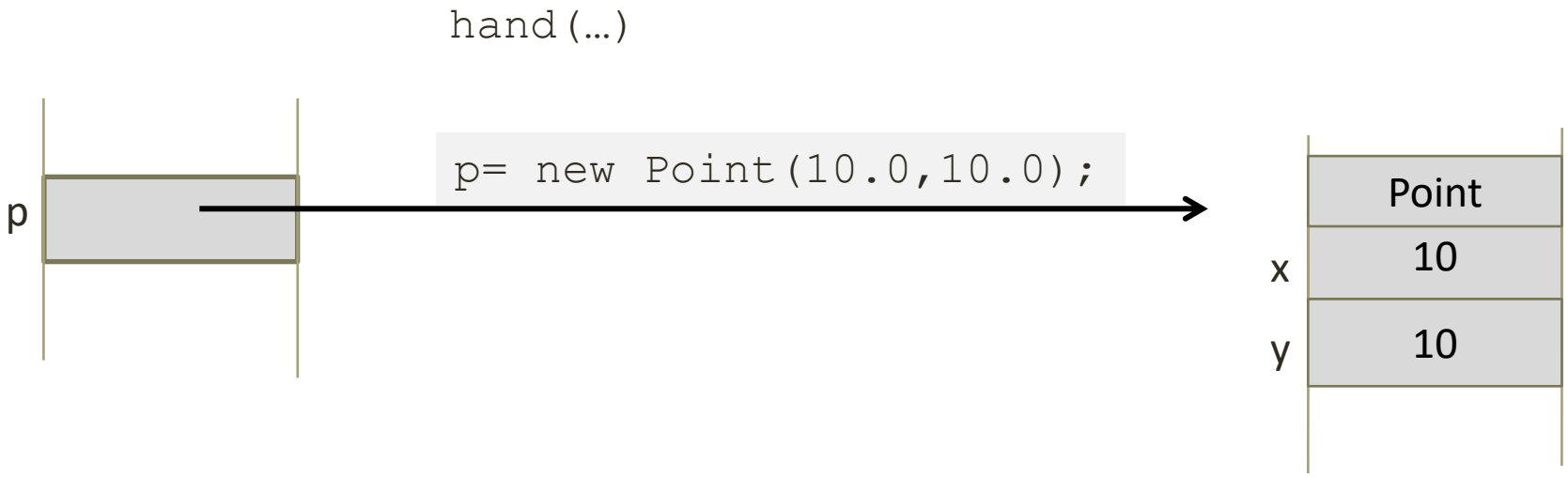
**Example**

```
public class Point{
  private double x;
  private double y;
public Point(double x, double y ){ this.x =x; this.y =y; }
public static void move ( Point pt, double dx, double dy ){
       pt.x = pt.x+dx ;
       pt.y = pt.y+dy ;
}
public String toString (){ return "Point(" + x +","+ y +")";}

 public static void main(String arg []){
Point p= new Point(10.0,10.0);
    System.out.println ("Before calling move " + p.toString ());
    move ( p,5.0,5.0);
    System.out.println (" After calling move "+ p.toString ());
}
}
```

Result displayed
Before calling move Point (10.0,10.0 )
After the call to move Point(15.0,15.0 )

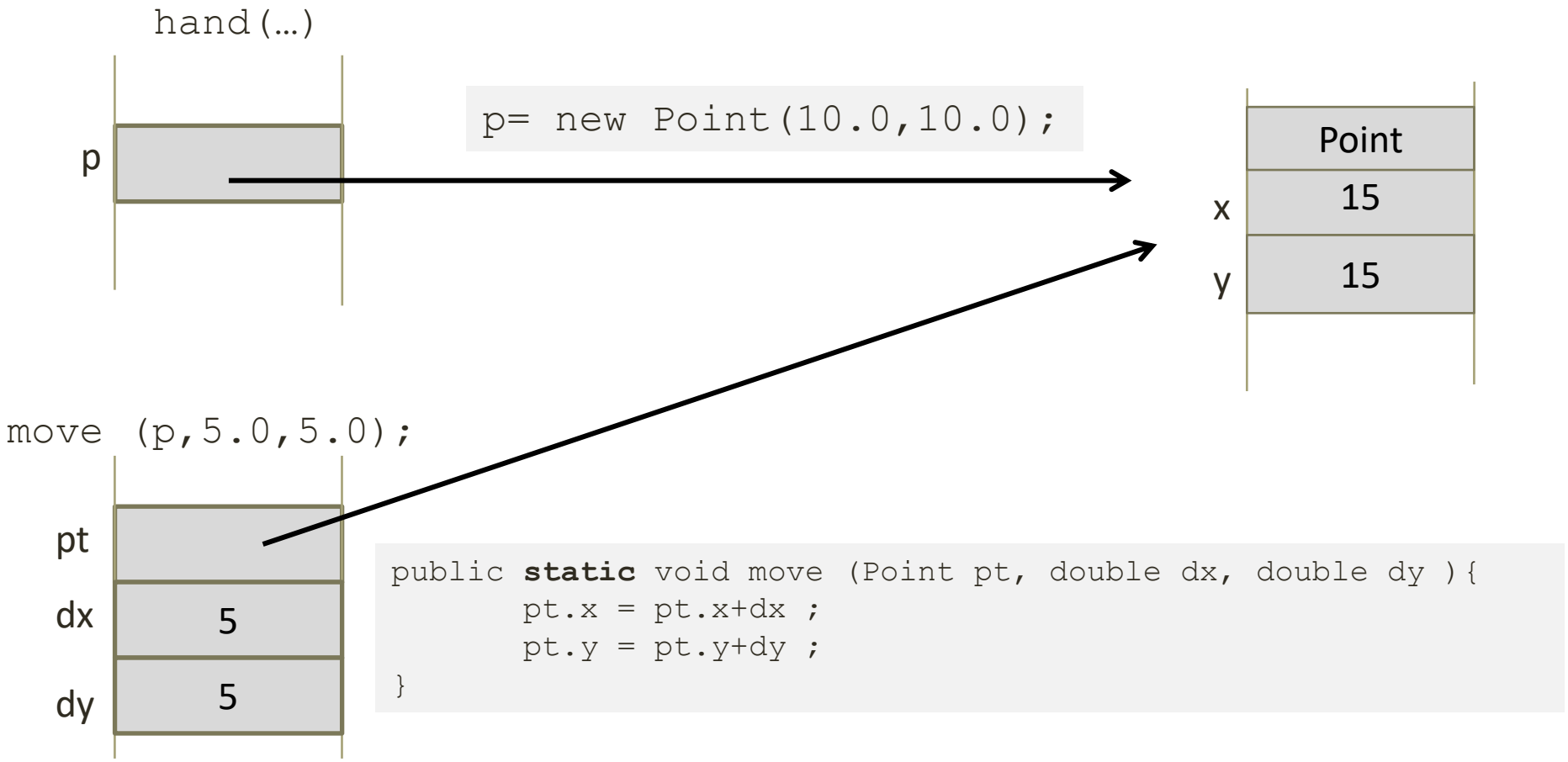
# 6. Passing parameters

**Example**



# 6. Passing parameters

**Example**




```
public static void move (Point pt, double dx, double dy ){
        pt.x = pt.x+dx ;
        pt.y = pt.y+dy ;
}
```

Result displayed
Before calling move Point (10.0,10.0 )
After the call to move Point(15.0,15.0 )

# 7. Static elements

# 7. Static elements

## 7.1 . Static attributes (class attributes)

- attributes are defined with the `static` keyword ;
- There is only one copy of the *static attribute* for all objects of the class;
- If a single object changes the value of a static attribute, its value will be changed for all objects of the class.
- To access a static attribute, we use the notation:

**NomClasse.nomAttribut**

**Example**

```
public class car
{
static byte nbRoues = 4;
private double length;
    private byte nbPassengers ;
}
```

# 7. Static elements

## 7.1 . Static attributes (class attributes)

- A classic usage of the static attribute is given by the following example:

### Example:

We wanted to add an identification attribute " **id** " to the `Person class`, such that each object of the `Person class` will have its own value for this attribute (no two objects should have the same value for the " **id** " attribute).

### Solution:

1. Declaration of the attribute " **id** " and a **static attribute** " **number** " initialized to 0.
2. In the constructor of the `Car class` : Assign the value of the " `number` " attribute to the " `id` " attribute, and increment the value of the "number" attribute.

# 7. Static elements

## 7.1 . Static attributes (class attributes)

- **Example (continued)**

```
public class Person {
//Attributes
 private int id;
public static int number=0 ;
 private String name ;
//Constructor
public Person(String name){
   id = number;
number++;
   this.name =name;
}
// Method toString ()
public String toString ()
{
return "Id:"+ this.id+", Name:"+ this.name ;
}
}
```

# 7. Static elements

```
public class MainClass {
public static void main(String arg []){

      Person p1=new Person("Ahmed");
Person p2=new Person("Ali");
Person p3=new Person("Aicha");

       System.out.println (p1.toString ());
   System.out.println (p2.toString());
  System.out.println (p3.toString());

       System.out.println    ("    Number    of    objects    ="+
Person.number );
}
}
```

**The displayed result is:**

```
Id:0, Name:Ahmed
Id:1, Name:Ali
Id:2, Name:Aicha
Number of objects=3
```

# 7. Static elements

## 7.2. Static methods

- The advantage of static methods is that they can be called when you don't have an object.
- A static method can only use static attributes and methods .
- To call a static method :

```
ClassName.MethodName ()
```

- **Example**

```
public class Adder
{
public static int sum( int a, int b)
{
return (a+b);
}
}
```

- To call the `sum` method , we will not need to create an object of the `Additionneur class` , we just need to write for example:

```
Adder.sum (5,10);
```

# 7. Static elements

## 7.2. Static methods

- The `main()` method is an example of static methods .
- It is the `main method` that is called when the `JVM` needs to execute a particular class.
- The `main()` method is static, so it is a method called by the class and not an object (No calling object).
  - To be able to call the methods of an object within the `main() method` , it is necessary to create an object of this class within the `main() method` .

**Example**

```
public class Person {
// Attributes
...
// Methods
...
audience  static void main(String[] args ) {
setName ( " Ali " );// error , main() is not executed by no
object
Person pers = new Person();
pers.setName ("Ali");// correct
}
```

# 8. Method overloading

# 8. Method

- Overloading is the process of defining multiple methods with the same name within the same class.
- Methods with the same name have *signatures* different,
- We call *signature* of a method the set consisting of the *name of the method* and the *parameters* passed to it.
- Two methods of objects of the same class that have the same name but do not have the same parameters, do not have the same signature and JAVA can distinguish them.
- The compiler chooses which method should be called based on the number and types of the parameters .
- Overloading allows you to simplify the interface of classes with respect to other classes .

# 8. Method

## Example

```
public class Adder{
public int sum( int a, int b) // 1
{return (a+b);}
public int sum( int a, int b, int c) // 2
 {return ( a+b+c );}
public float sum (float a, float b) // 3
 {return (a+b);}
 public float sum( int a, int b) //4
 {return ((float)a+(float)b); } //error
}
```

Method 4 declaration causes an error because it has the same signature as method 1.