Mila University Center
2nd Year – Bachelor's in Computer Science
Course: Object-Oriented Programming

CHAPTER II:

Class and Object

Lecturer : DR. SADEK BENHAMMADA
EMAIL : s.benhammada@centre-univ-mila.dz

1. Class declaration syntax

1. Class declaration syntax

Header

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

Body

```
 {
  //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 Primitive 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. Method Declaration: The declaration of a method is composed of the **signature** and the **body :**

- **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. Primitive types**

Types	Size	values	Example
byte	1 byte	Integers between -128 and +127	byte temperature ; temperature = 64;
shorts	2 bytes	Integers between -32768 and +32767	short speedMax ; speedMax = 32000;
int	4 bytes	Integers between -2147483648 and 2147483647	int temperatureSun ; temperatureSun = 15600000;
long	8 bytes	Integers between - 9223372036854775808 and 9223372036854775807	long yearLight ; lightyear =9460700000000000;

float	4 bytes	Floating point numbers between 1.401e-045 and 3.40282e+038	float pi;
			pi = 3.141592653f ;

double	8 bytes	Floating point numbers between 2.22507e-308 and 1.79769e+308	double division ; division = 0.333333333333334 ;
tank	2 bytes	character (65000 characters possible)	char character ; character = 'A'

boolean	1 bit	logical value: true or false	boolean question;
			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

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

```
public 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. Java Naming Conventions

4. When a name is made up of several words :

- **Class and Interfaces**: Use PascalCase (capitalize the first letter of each word).

  Example:

  ```
  public class BankAccount {…} //Class
  ```

- **Attributes and methods**: Use camelCase (start with a lowercase letter, capitalize subsequent words).

Examples :

```
public int numberWheels ; //attribute
public double calculateArea(){…}; // method
```

5. **Constant** should be written in all UPPERCASE letters with underscores separating words.

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 , describing the action they perform.

   Example :

   ```
   public double calculateArea ( )
   ```

7. It is common for all names to be in English.

3. Object Declaration and Creation

3. Object Declaration and Creation

3.1. Declaring an object

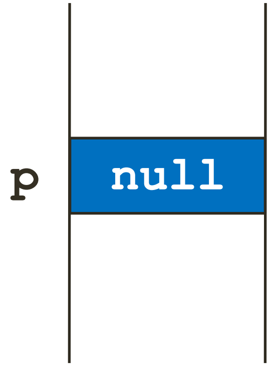
- The declaration of an object is of the form:

```
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 stage, the variable **p** does not refer to any actual object in memory. It simply reserves a memory location to hold a reference to a Point object.

3. Object Declaration and Creation

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. Object Declaration and Creation

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.

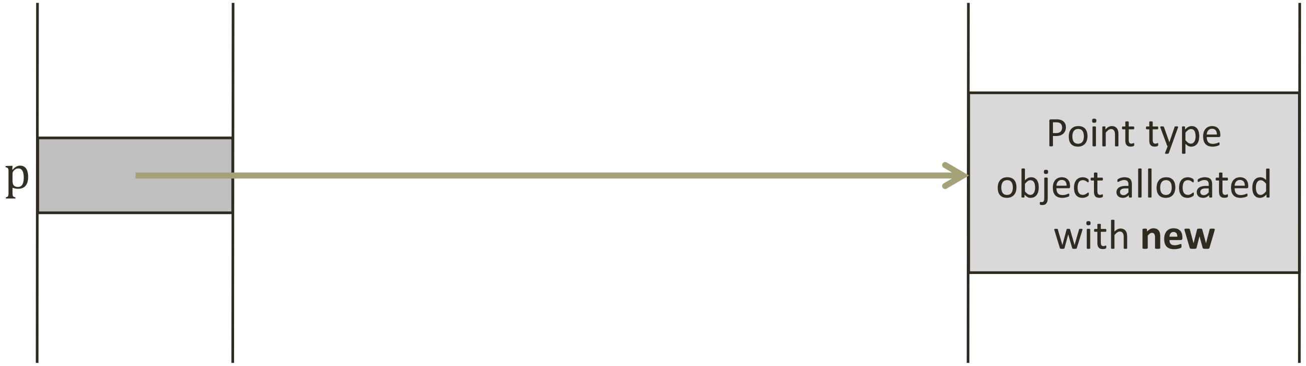
Example

```
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. Object Declaration and Creation

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. Object Declaration and Creation

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. Object Declaration and Creation

The this Keyword

- 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. Object Declaration and Creation

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 method parameters have the same name as attributes.
- **Example :**

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

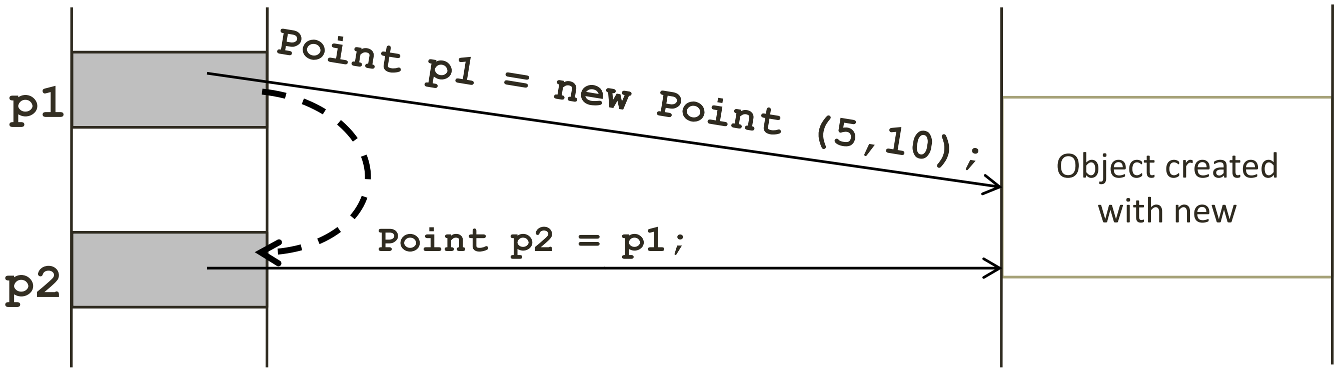
```
//  Constructor of the Point class

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

3. Object Declaration and Creation

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. Object Declaration and Creation

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`

```
public class Point {
  private double x ;
  private double y ;
     // Constructor
  public 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 ;
}

 }
```

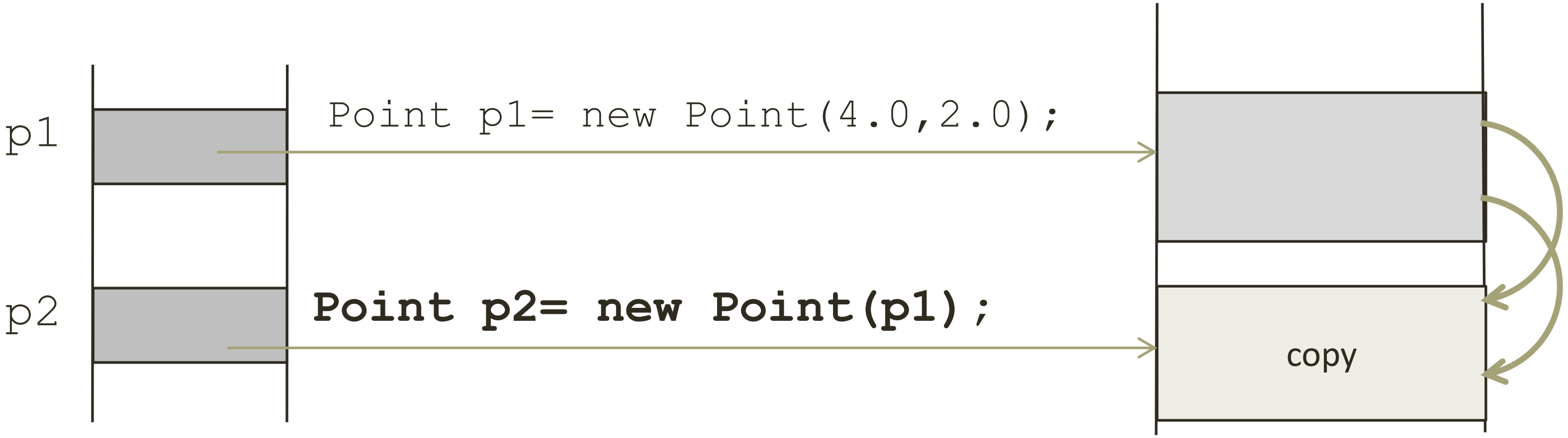
3. Object Declaration and Creation

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. Object Declaration and Creation

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 :

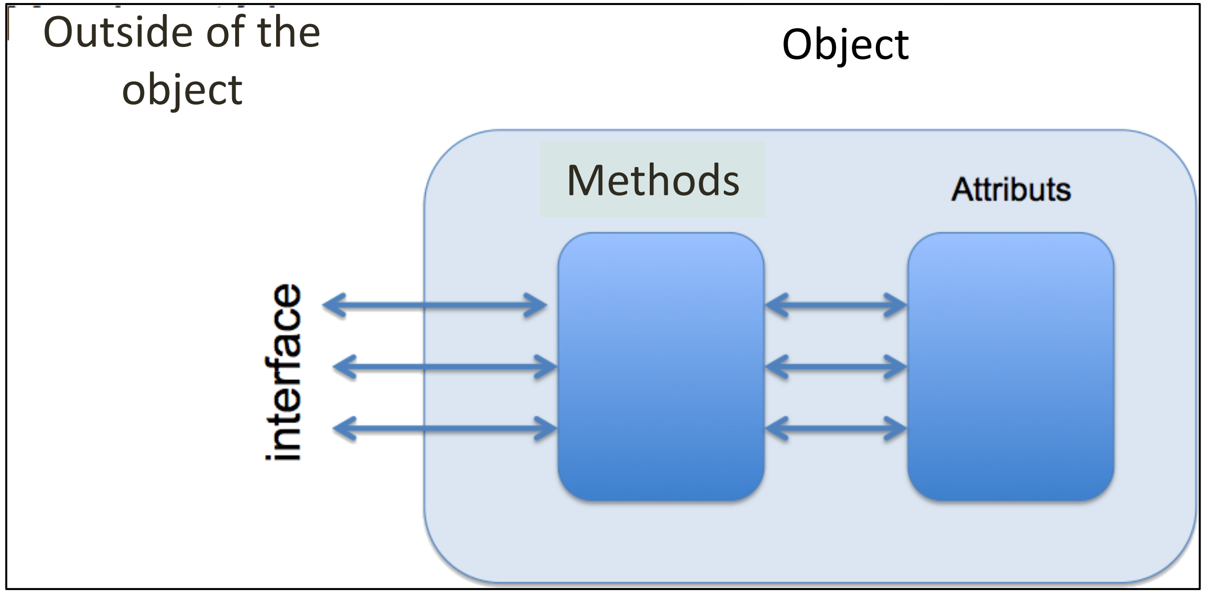
1. The declaration and creation of the object.
2. Using the object by calling these methods.
3. Object deletion: it is automatic in Java thanks to the memory collector (`Garbage Collector :` **GC**).

- 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 .
  - *Public* : 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

- **Attributes** of a class are typically declared **private** (private) or **protected** (protected), meaning they are **not directly accessible** from outside the class. This ensures **data encapsulation** and **prevents unintended modifications**.
- **Methods** are usually declared **public** (public), meaning any object can call them.
- Example

```
public class Person {
    private String name;  // Private attribute (not directly accessible)
    protected int age;    // Protected attribute (accessible in subclasses)

    // Public method (accessible everywhere)
    public void setName(String name) {
        this.name = name;
    }

    // Public method (getter)
    public String getName() {
        return name;
    }
}
```

4. Encapsulation

4.2. Reading and Modification (Accessors)

- To **read and modify** the attributes of an object while maintaining **encapsulation**, we use specially designed methods called **accessors**.
- These accessors ensure **controlled access** to private attributes, preventing direct modification from outside the class.

Reading Accessors (Getters)

- **Getters** are methods that allow reading (retrieving) an object's private attributes.
- The method name typically starts with **"get"** followed by the attribute name (in camel case).
- Getters return the value of the attribute but do not modify it.

Example

```java
public class Person {
    private String name;  // Private attribute

    // Getter method to retrieve the name
    public String getName() {
        return name;
    }
}
```

4. Encapsulation

4.2. Reading and Modification (Accessors)

- **Modification Accessors (Setters)**
- **Setters** are methods that allow modifying (updating) an object's private attributes.
- The method name typically starts with **"set"** followed by the attribute name.
- Setters take a parameter and assign it to the private attribute.
- **Example**

```
public class Person {
    private String name;  // Private attribute

    // Setter method to modify the name
    public void setName(String n) {
        name = n;
    }
}
```

4. Encapsulation

Example : Point Class (Encapsulation with Getters and Setters)

```
public class Point {
    // Attributes (private for encapsulation)
    private double x;
    private double y;

    // Getter methods (read accessors)
    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    // Setter methods (modification accessors)
    public void setX(double x) {
        this.x = x;
    }

    public void setY(double y) {
        this.y = y;
    }
}
```

4. Encapsulation

Example : MainClass (Testing the Point Class)

```java
public class MainClass {
    public static void main(String[] args) {
        // Creating a Point object using the default constructor
        Point p = new Point();

        // Using setters to modify attributes
        p.setX(5.0);
        p.setY(10.0);

        // Using getters to read and display attribute values
        System.out.println("X coordinate: " + p.getX());
        System.out.println("Y coordinate: " + p.getY());
    }
}
```

The result displayed:

```
X coordinate: 5.0
Y coordinate: 10.0
```

4. Encapsulation

4.3. Importance of Accessors (Getters and Setters)

The main advantage of using **accessors (getters and setters)** is that they make the rest of the code **independent of the internal representation of an object**.

1. Encapsulation & Data Protection

- Attributes are **kept private** (private) and can only be accessed or modified through methods, ensuring **better control** over data.
- Prevents **accidental modifications** or **direct manipulation** of sensitive data.

2. Flexibility & Maintainability

- If we decide to **change an attribute's implementation**, we only **modify the getter or setter**, without affecting the rest of the code.
- Without accessors, every part of the program that uses the attribute would need to be modified, making maintenance **difficult and error-prone**.

4. Encapsulation

- **4.3. Importance of Accessors (Getters and Setters)**

Example : Direct Access (Not Recommended)

```java
public class BankAccount {

// Direct modification (Unsafe)
      public double balance;
}
```

```java
// MainClass
public class MainClass {
         public static void main(String[] args) {
                  BankAccount account = new BankAccount();
// Direct modification (Unsafe)
                  account.balance = 500;
                  System.out.println("Balance: " + account.balance);
         }
}
```

Problem: If we later decide to add validation (e.g., no negative balances), we must modify **every line** that directly accesses balance.

4. Encapsulation

- **4.3. Importance of Accessors (Getters and Setters)**

Example : Using Getters and Setters (Best Practice)

```
public class BankAccount {
private double balance; // Private attribute
(Encapsulation)
   // Getter method (Read access)
   public double getBalance() {
      return balance;
   }
   // Setter method (Write access with validation)
   public void setBalance(double balance) {
      if (balance >= 0) {
        this.balance = balance;
      } else {
        System.out.println("Balance cannot be negative!");
      }
   }
}
```

```
// MainClass
public class MainClass {
   public static void main(String[] args) {
     BankAccount account = new BankAccount();
     account.setBalance(500); // Using setter
     System.out.println("Balance: "+account.getBalance());
     //  Using getter
     account.setBalance(-100); // Balance cannot be
     negative!
   }
}
```

5. Packages

5. Packages

5.1. Definition

- A **package** is a collection of related **classes, interfaces, and sub-packages** that are grouped together under a common name.

Importance of Using Packages:

- **Improves Code Organization**: Groups similar classes together, making projects structured and manageable.
- **Enhances Code Reusability**: Packages allow modular design, making it easier to reuse and import code.
- **Access Control & Encapsulation**: Provides better control over class visibility

5. Packages

5.2 Declaring a Package

- A package is declared at the top of a Java file using the package keyword.
- By convention, the name of a package begins with a lowercase letter.
- **Syntax:**

```
package packageName ;
```

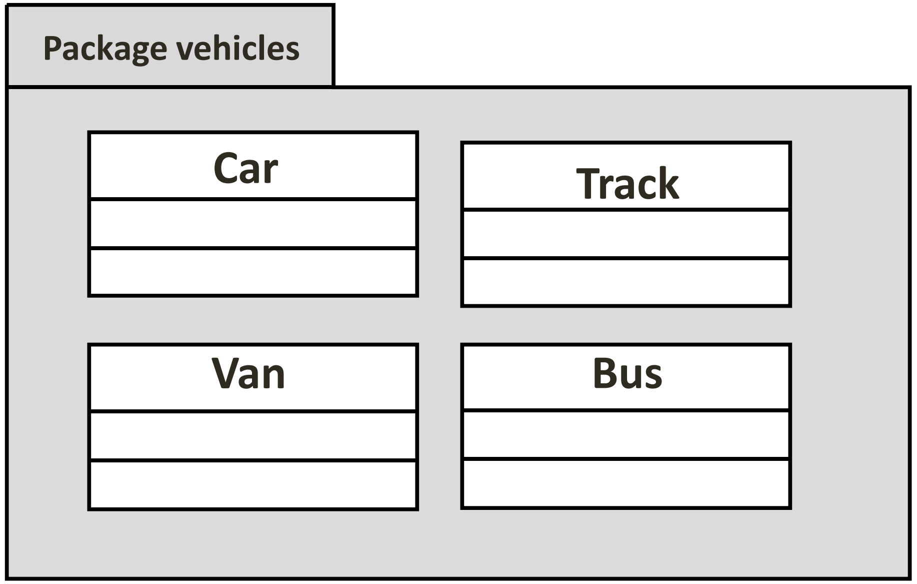
Example

```
package vehicles;
public class Car{...}
```

```
package vehicles;
public class Track {...}
```

```
package vehicles ;
public class Van {...}
```

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



5. Packages

5.3. Class Access Control

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

1. **Public Class** : The class is visible to classes in its package , and outside the package .
   - **Syntax:**

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

Example

```
public class Car {...}
```

2. **Package-Private Class (Default Visibility):** The class is only visible to classes only accessible within the same package. It **cannot** be accessed from **another package**, even if imported.
   - **Syntax:**

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

Example

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

5. Packages

5.3 . Using a package

- When referencing a **class from another package**, there are two ways to access it:

1. Importing the class : The recommended approach is to use the import statement to import the class before using it.

- **Syntax**

```
import PackageName.ClassName ;
```

2. Using the Fully Qualified Name : Precede each occurrence of the class name with the name of the package in which it is defined

5. Packages

Example

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

Importing the class :

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

Using the Fully Qualified Name :

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

5. Packages

- For import all classes from a package:

```
import nomPackage .*;
```

Example

```
package vehicles;
public class Car{...}
```

```
package vehicles ;
public class Van {...}
```

```
package vehicles;
public class Track {...}
```

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

The following code :

```
Import vehivles.Car;
Import vehivles.Track;
Import vehivles.Van;
Import vehivles.Bus;
```

can be replaced by the following code:

```
import vehicle.*
```

5. Packages

5.4 Creating Sub-Packages

- A **sub-package** is a package inside another package. It helps in better organization of related classes.

Example: Creating a Sub-Package vehicles.cars

```
package vehicles.cars;  // Declaring sub-package cars of package vehicule
public class SportsCar {
    ...
}
```

Importing a Class from a Sub-Package java

```
import vehicles.cars.SportsCar;  // Importing from sub-package
public class Main {
    public static void main(String[] args) {
        SportsCar ferrari = new SportsCar();
        ...
    }
}
```

5. Packages

5.5 Default Package (No Package Declaration)

- If a Java file **does not specify a package**, it is placed in the **default package** (not recommended for large projects).
- **Example**

```
public class DefaultClass {
    public void display() {
        System.out.println("This class is in the default package.");
    }
}
```

- **Limitation:** Classes in the default package cannot be imported in files that belong to a named package.

6. Parameter Passing in Methods

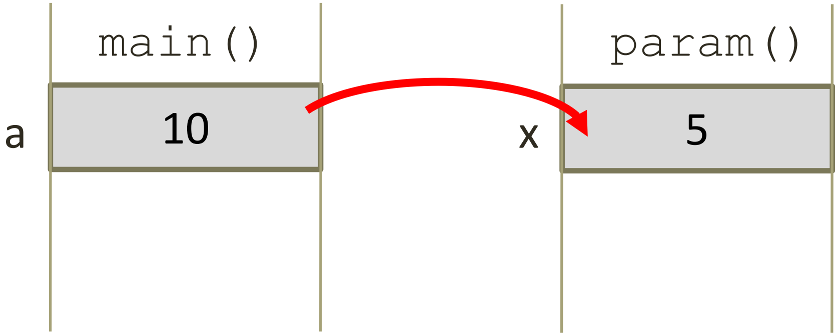
6. Parameter Passing in Methods

- In Java, **parameters are always passed by value**, meaning that the **value of the actual parameter** is **copied** into the corresponding **formal parameter** when a method is called.
  - When each method is called, local memory space is allocated for each **formal parameter**;
  - The values of the **actual parameters** (arguments) are **copied** into the corresponding **formal parameters** before execution.
  - The method **works on the copied values**, not on the original arguments.
  - Changes made inside the method do not affect the original variables (for primitive types)

6. Parameter Passing in Methods

- **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. Parameter Passing in Methods

Passing an Object as a Parameter in Java

- In Java, **when an object is passed as a parameter to a method, it is the reference to the object that is passed and copied into the formal parameter**.

- The **memory address (reference)** of the object is copied, **not the object itself**.
- Since both the actual parameter (original object) and the formal parameter (method argument) **point to the same object in memory**, **modifications made inside the method affect the original object**.

6. Parameter Passing in Methods

Example

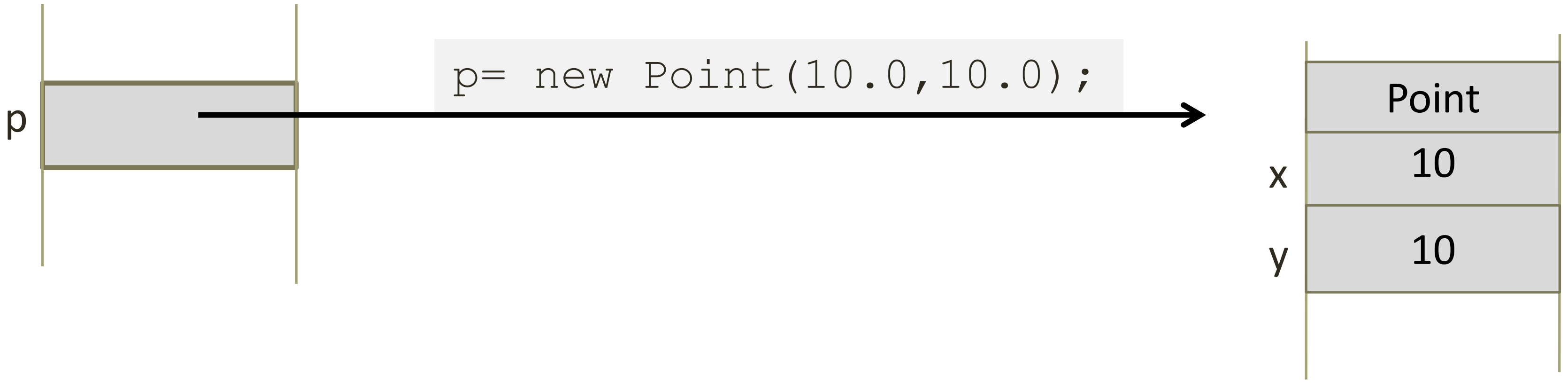
```
public class Point{
    private double x;
    private double y;
    public Point(double x, double y ){ this.x =x; this.y =y; }
    public static void move ( Point pt, double dx, double dy ){
        pt.x = pt.x+dx ;
        pt.y = pt.y+dy ;
    }
    public String toString(){ return "Point(" + x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

Example (explanation)

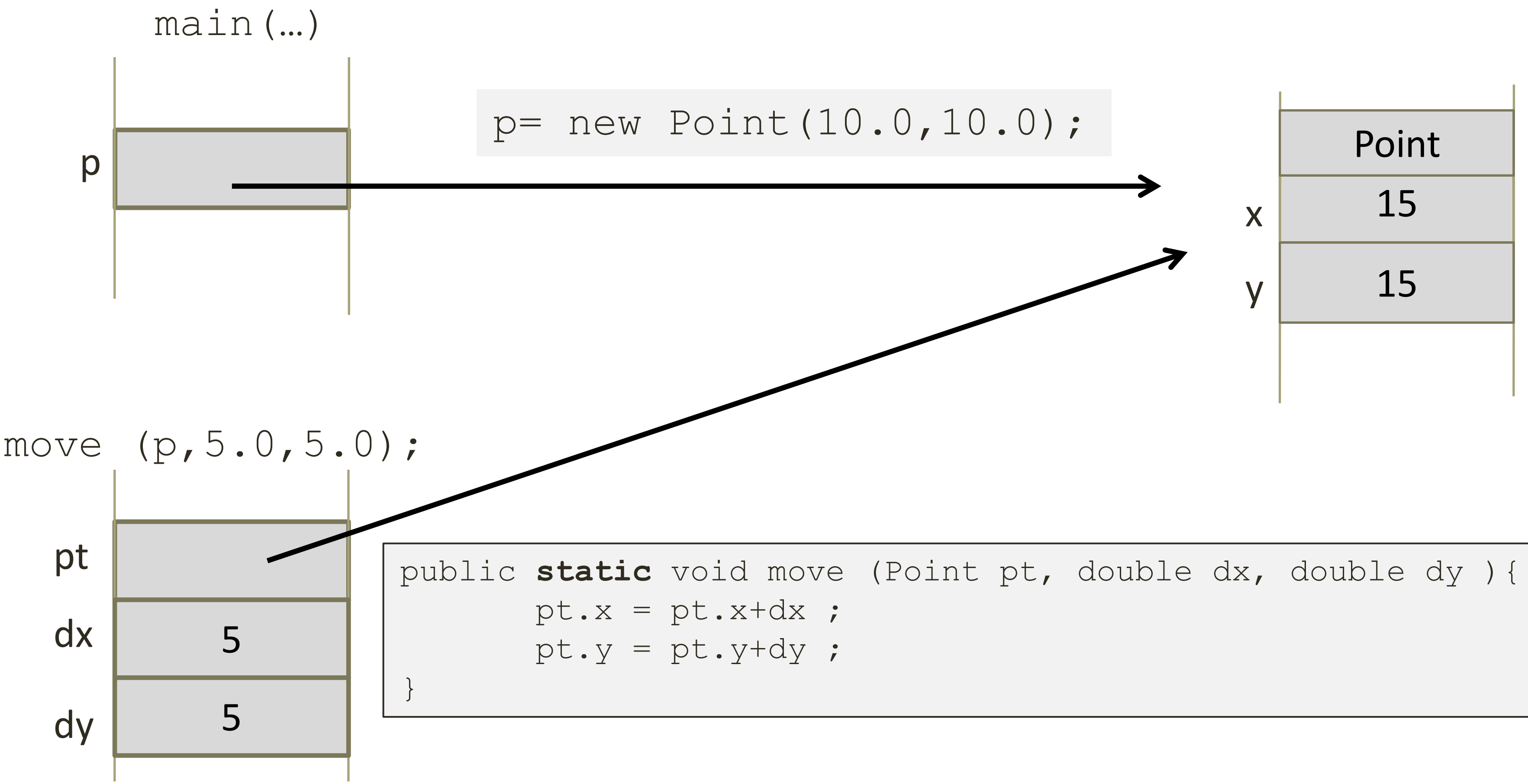
```
main(…)
```



```
p= new Point(10.0,10.0);
```

6. Parameter Passing in Methods

Example (explanation)



```
p= new Point(10.0,10.0);
```

```
move (p,5.0,5.0);
```

```
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

- Static elements in Java belong to the **class itself rather than instances (objects)** of the class.
- This means they are **shared across all objects** and do not require object instantiation to be accessed.
- There are **two main types** of static elements in Java:

  1. **Static Attributes (Class Variables)**
  2. **Static Methods (Class Methods)**

7. Static elements

7.1 . Static attributes (class attributes)

- Static 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 wheelCount = 4;
    private double length;
    private byte nbPassengers ;
}
```

7. Static elements

7.1 . Static 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**

Example

```java
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 ;
    }
}
```

```java
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:

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

7. Static elements

7.2. Static methods

- A static method belongs to the class rather than an instance.
  - Called using the class name (no need for an object).
  - Cannot access non-static attributes or methods directly.
  - A static method can only access static attributes and methods .

- To call a static method :

```
ClassName.methodName()
```

- The advantage of static methods is that they can be called when you don't have an object.
- **Example**

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

```
public class Main {
    public static void main(String[] args) {
        int sum = Adder.add(5, 10);
        System.out.println("Sum: " + sum); // Output: Sum: 15
    }
}
```

- To call the `sum` method , we will not need to create an object of the *Adder* 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
      ...
    public  static void main(String[] args ) {
        //Incorrect: setName() is an instance method, but no object exists
        setName("Ali"); // ERROR: Cannot call non-static method from a static context
        //Correct: Create an instance of Person before calling setName()
        Person pers = new Person();
        pers.setName ("Ali");// correct
    }
}
```

8. Method overloading

8. Method overloading

- **Polymorphism** allows **one interface to have multiple implementations**, making the code more flexible and scalable.

Types of Polymorphism:

1. **Method Overloading:** Multiple methods with the **same name but different parameters** in the same class. ***(Covered in This Section)***

2. **Method Overriding:** A subclass **redefines** a method inherited from the parent class (explained in chapter 3). ***(Covered in The next Chapter)***

8. Method overloading

- **Method overloading** is the process of defining **multiple methods with the same name** within the **same class**, but with **different parameter lists**.
- Each method has a **unique signature**, which consists of:
  - Method ***Name***
  - ***Number***, ***Type***, and ***Order*** of Parameters
- If two methods have the **same name** but **different parameters**, they are considered **overloaded methods**.

- Java **differentiates** between overloaded methods based on their **signatures** (method name + parameters).

- Overloading **simplifies class design**, making it more intuitive and flexible by allowing multiple ways to use a method.

8. Method overloading

Example

```
public class Adder {
   // Method 1: Sum of two integers
   public int sum(int a, int b) {
       return a + b;     }

   // Method 2: Sum of three integers (Overloaded)
   public int sum(int a, int b, int c) {
       return a + b + c;
   }

   // Method 3: Sum of two floating-point numbers (Overloaded)
   public float sum(float a, float b) {
       return a + b;
   }

   // Method 4: Overloading by Parameter Order (int, float)
  public float sum(int a, float b) {
     return a + b;
   }
```

```
   // Method 5: Overloading by Parameter Order (int, float)
   public float sum(float a, int b) {
      return a + b;
    }

// Method 6 : Compilation Error: Only return type is different (Duplicate
signature)
    /*
   public float sum(int a, int b) {
        return (float) (a + b);
   } /*

    // Method 6 (Fixed): Avoids duplicate signature issue by changing the
name methos
    public float sumAsFloat(int a, int b) {
        return (float) (a + b);
    }
}
```