3TC36: Object-Oriented Programming in Java

Visibility, scope of declarations, and data encapsulation

Dominique Blouin
dominique.blouin@telecom-paris.fr
Le February 20, 2026

Learning objectives

- Visibility of declarations.
- Data encapsulation.
- Scope of declarations.
- The `String` and `ArrayList` classes.

Recap

- **Objects** are computational entities that communicate by sending messages.
- Objects contain values called **attributes**. Among the attributes, we can find **references** to other objects.
- A reference to an object allows sending it a **message**.
- For each type of message an object can receive, it knows a **method** associated with that message type.
- This method is a procedure **executed by the object** when it receives the associated message type.

Recap

- An object type is described by a **class**.
  - The class describes the **attributes**: name and value type.
  - The class describes the **methods** used to respond to messages.
- The programmer can create **objects** from the class. This is called **instantiation**.
- We say that objects are **instances** of the class or that objects **belong** to the class.

Delegation and non-intrusion

- A good practice in OOP is to avoid acting from the outside on an object's state (**non-intrusion**).
- Thus, we will avoid direct *read* and *write* operations on attributes as much as possible.
- In our example with the `Point` class, we will ask the object of type `Point` to display its coordinates (**delegation** principle).
- As seen in the previous class:

```
Point myPoint = new Point(10, 10);
myPoint.writeCoordinates();
```

Good practice

- Example: Display the coordinates of a point.

```
class Point {
  int xCoord;
  int yCoord;
  void writeCoordinates() {
      System.out.print("x = ");
      System.out.println(xCoord);
      System.out.print("y = ");
      System.out.println(yCoord);
  }
}
```

- Reminder: To access an attribute or a method of an object, we use the "." character.

```
Point myPoint = new Point(10, 10);
myPoint.writeCoordinates();
```

Bad practice

- Direct access to attributes:

```
Point myPoint = new Point(10, 10);
System.out.print("x = ");
System.out.println(myPoint.xCoord);
System.out.print("y = ");
System.out.println(myPoint.yCoord);
```

- How **to prevent** this bad practice in Java?
- Answer: Use Java's **visibility** mechanism.

Visibility in Java

- Visibility determines which elements of a class (attributes, methods, or classes) are accessible by other classes.
- Four types of visibility:
  - `public` : Everything is accessible.
  - `private` : Accessible only within the **class**.
  - `package` (default if nothing is specified): Accessible only within classes of the same **package**.
  - `protected` : Accessible only within **inheritance** classes (concept to be discussed in the next lesson) or the same **package**.

`private` visibility

- Example of `private` visibility:
- Using **private** visibility prevents accessing the attributes from **outside** the class.
  - Compilation **errors** occur.
- Inside the class, such as in the `writeCoordinates()` method, the attributes are always accessible.
- Visibility can also apply to **methods**, including **constructors**.
- The `writeCoordinates()` method will be declared as `public` to make it accessible to all classes.

```
class Point {
  private int xCoord;
  private int yCoord;
}
```

```
Point myPoint = new Point(10, 10);

System.out.print("x = ");
System.out.println(myPoint.xCoord);
System.out.print("y = ");
System.out.println(myPoint.yCoord);
```

```
void writeCoordinates() {
    System.out.print("x = ");
    System.out.println(xCoord);
    System.out.print("y = ");
    System.out.println(yCoord);
}
```

Data encapsulation

- An object's attributes hold values that define its **state**.
- If the attributes of an object are declared with the `private` keyword, then only the object itself can access them.
- This is called **data encapsulation** within the class.

```
class Point {
    private int xCoord;
    private int yCoord;
}
```

Data encapsulation

- Data encapsulation is **fundamental** in OOP.
- If data is not encapsulated, any other object can modify it, which can lead to inconsistencies.
- Data encapsulation involves two key aspects:
  - **Accessing** the data.
  - **Modifying** the data.

Data access (Getters)

- If we want **to know** the value of an object's attribute, we need **to ask** the object.
- To do this, we define a specific method called a **getter** (since the method name usually starts with `get`).
- A getter can also be called an **accessor** or **access method**.
- The getter is a **public method** that returns the value of a private attribute.

Examples of accessor declarations

```
class Point {
    private int xCoord; // The pattern: private attribute
    private int yCoord; // The pattern: private attribute

    public Point(int xCoord, int yCoord) {
        this.xCoord = xCoord;
        this.yCoord = yCoord;
    }
    public int getxCoord() {   // The pattern: public getter
        return xCoord;
    }
    public int getyCoord() {   // The pattern: public getter
        return yCoord;
    }
    //...
}
```

Examples of using accessors

```
Point myPoint = new Point(5, 7);
int xCoord = myPoint.getxCoord();
```

Modifying data (Setters)

- To modify the value of an object's attribute, we need **to ask** the object.
- To do this, we define a specific method called a **setter** (since the method name usually starts with `set`).
- A setter can also be called a **mutator** or **modification method**.
- The setter takes a parameter (the new value) and assigns it to the private attribute.

Examples of mutator declarations

```
class Point {

    private int xCoord; // The pattern: private attribute
    private int yCoord; // The pattern: private attribute

    public void setxCoord(int xCoord) {   // The pattern: public setter
        this.xCoord = xCoord;
    }

    public void setyCoord(int yCoord) {   // The pattern: public setter
        this.yCoord = yCoord;
    }
}
```

Examples of mutator declarations

```
Point myPoint = new Point(5, 7);
myPoint.setxCoord(9);
```

Why encapsulate?

- Data encapsulation is a **very important** principle.
- All attributes must be qualified with the `private` keyword, meaning that attributes can only be accessed or modified from within the object's **methods**.
- There are multiple reasons for this principle. Let us look at a few.

Reason 1: Validate attribute values

- Having **setters** allows us **to control the validity** of the values we want to assign to attributes.

```
class Person {

    private int age;

    public boolean setAge(int age) {
        if (age >= 0 && age < 150) {
            this.age = age;
            return true;
        }
        return false;
    }
    //...
}
```

Reason 2: Data consistency

- Encapsulation helps preserve the consistency and integrity of data structures. Example:
- The `length` attribute is always updated / recalculated whenever the `dx` or `dy` attributes are changed, *i.e.,* all three values are consistent.

```
class Vector {

    private int dx;
    private int dy;

    private double length;

    public Vector(int dx,
                  int dy) {
        setDxDy(dx,dy);
    }

    public void setDxDy(int dx, int dy) {
        this.dx = dx;
        this.dy = dy;
        length = Math.sqrt(dx*dx + dy*dy); // Maintain
        data consistency
    }

    public double getLenth() {
        return length;
    }
}
```

Reason 3: Application consistency

- Data encapsulation helps guarantee the **consistency** of an application.
- Let us assume that geometric figure classes, such as the `Point` class, are being edited in an editor:
  - Any modification of the characteristics of a figure, such as its position, must trigger a **refresh** of the display in the editing window.
  - Otherwise, the application will be inconsistent: the view (display) will not reflect the model (data).
- The attribute modification methods (mutators or setters) will be responsible for triggering a refresh of the editing window.
- This will be explored in more detail in the course on the **MVC** (Model View Controller) design pattern.

Reason 4: Implementation transparency

- The methods of a class that allow communication with objects of that class form the **interface** of the object with the outside world (other objects).
- These methods must be visible to all objects and will be marked with the `public` keyword.
- The attributes are hidden and marked with the `private` keyword.
- If data is correctly encapsulated, it is possible to modify the implementation of the methods **transparently**.

Example of implementation transparency (1/3)

- Previous example: Encapsulation helps preserve the consistency and integrity of data structures.

```
class Vector {

    private int dx;
    private int dy;

    private double length;

    public Vector(int dx, int dy) {
        setDxDy(dx,dy);
    }

    public void setDxDy(int dx, int dy) {
        this.dx = dx;
        this.dy = dy;
        length = Math.sqrt(dx*dx + dy*dy);
    }

    public double getLenth() {
        return length;
    }
}
```

Example of implementation transparency (2/3)

- Suppose we **rarely** need to know the length of a vector.
- Then we can implement the class differently, **transparently**:
  - The interfaces of both classes are the **same**.

```
class Vector {

    private int dx;
    private int dy;

    private double length;

    public Vector(int dx, int dy) {
        setDxDy(dx,dy);
    }

    public void setDxDy(int dx, int dy) {
        this.dx = dx;
        this.dy = dy;
        length = Math.sqrt(dx*dx + dy*dy);
    }

    public double getLenth() {
        return length;
    }
}
```

Example of implementation transparency (3/3)

- Suppose we **rarely** need to know the length of a vector.
- Then we can implement the class differently, **transparently**:
  - The interfaces of both classes are the **same**.

```
class Vector {

    private int dx;
    private int dy;

    public Vector(int dx, int dy) {
        setDxDy(dx,dy);
    }

    public void setDxDy(int dx, int dy) {
        this.dx = dx;
        this.dy = dy;
    }

    public double getLenth() {
        return Math.sqrt(dx*dx + dy*dy);
    }
}
```

Implementation transparency through data encapsulation

- Because the data is well encapsulated in the `Vector` class, either implementation of the class can be used **transparently**.
- Both classes implementing the `Vector` class have the **same interface**, meaning they present the **same methods** to the objects that use them.
- If the data is properly encapsulated in a class, the class user does not need to know **how the data is structured** or **how the methods are implemented** inside the class objects.

Packages: Organizing classes

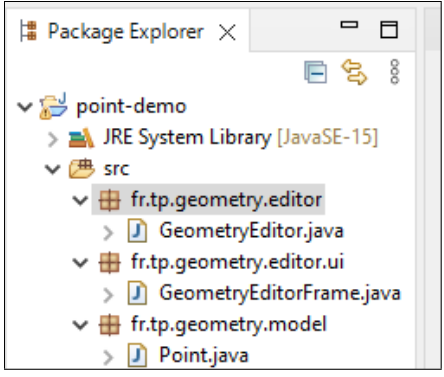
- A **package** is a **logical** collection of classes.
- In a Java program, classes are grouped into **packages**:
  - A package for input/output (I/O) classes.
  - A package for network communication classes.
  - A package for graphical user interface (UI) classes.
  - Etc.
- Principle of **strong cohesion** and **weak coupling**:
  - Classes that are highly dependent on **each other** (strong coupling) should be grouped in the **same package** to maintain strong cohesion.
  - Classes with few dependencies between them and related to **different functionalities** (weak coupling) will be grouped in **different packages**.
  - Goal: to facilitate the **reuse** of classes.

Package declaration (1/2)

```java
package fr.tp.geometry.model;

class Point {
    private int xCoord;
    private int yCoord;

    public Point(int xCoord,
         int yCoord) {
        this.xCoord = xCoord;
        this.yCoord = yCoord;
    } // ...
}
```



Package declaration (2/2)

- Packages are **hierarchical**:
  - A package can contain other packages.
- The class file (`Point.java`) will be stored in a subdirectory that follows the **same structure** as the one defined by the **package name**.
  - The dots (`.`) are replaced by slashes (`/`).
  - Example path: `../src/fr/tp/geometry/model/Point.java`.
- Naming conventions:
  - Package names contain **only lowercase letters**.
  - The company's **domain name** is often used **to uniquely** identify the package.

Package namespace

- By default, a class can only access another class declared in a different package by specifying **the full package name**.

```
package fr.tp.geometry.editor;

public class Main {

  public Main() {
  } // This default constructor is not really needed to be explicit

  public static void main(String[] args) {
      fr.tp.geometry.model.Point myPoint =
        new fr.tp.geometry.model.Point(10, 10);

      myPoint.writeCoordinates();
  }
}
```

Package importing

- Specifying the full package name makes the code difficult to read.
- To solve this problem, we can use the `import` keyword:

```
package fr.tp.geometry.editor;
import fr.tp.geometry.model.Point;

public class Main {

  public Main() {
  } // This default constructor is not really needed to be explicit

  public static void main(String[] args) {
      Point myPoint = new Point(10, 10);

      myPoint.writeCoordinates();
  }
}
```

Importing all classes from a package

```
package fr.tp.geometry.editor;

import fr.tp.geometry.model.*; // "*" imports all the classes from the
    package

public class Main {

  public Main() {
  } // This default constructor is not really needed to be explicit

  public static void main(String[] args) {
      Point myPoint = new Point(10, 10);

      myPoint.writeCoordinates();
  }
}
```

`Package` type visibility (1/2)

- If no visibility is declared, the visibility will be that of the `package`.
- A class will only be able to access a `package`-visible element from another class if both classes are declared **in the same package**.
- If the classes are in two different packages, only elements declared with `public` visibility will be accessible.

Package type visibility (2/2)

```java
package fr.tp.geometry.model;

class Point {
    // Package visibility
    int xCoord;
    int yCoord;

    public Point(int xCoord,
                 int yCoord) {
        this.xCoord = xCoord;
        this.yCoord = yCoord;
    }
    // ...
}
```

```java
package fr.tp.geometry.model;

class Circle {
    private Point location;
    private float radius;

    public Circle(int xCoord,
                  int yCoord,
                  float radius) {
        this.location = new Point(xCoord, yCoord);
        this.radius = radius;
        this.location.xCoord = 0; // Accessible
    } // ...
}
```

Class visibility

- We can also define visibility for a class:

```
public class Vector {
    private int dx;
    private int dy;
    public void setDxDy(int dx, int dy) {
        this.dx = dx;
        this.dy = dy;
    }
    public double getLength() {
        return Math.sqrt(dx*dx + dy*dy);
    }
}
```

- Question: What is the meaning of class visibility?
- Answer: **Class visibility** controls **who can use (instantiate, extend, or reference) the class itself**, not its members.

Scope of declarations: Three kinds of variable declarations

```java
public class Test {
    private int myAttribute; // Attribute declaration

    public void myMethod(int myPar) { // Parameter declaration
        // ...
        int myLocalVar = 0; // Local variable declaration
        // ...
    }
}
```

- These declarations have a **scope**.
- Scope: the part of the program where the declaration is **visible**.
  - The **places in the code** where the variable can be **used**.

Scope of declarations

- Local variable:
  - The scope is **within the instruction block** starting from the line of the declaration of the variable.

```
{    // ... ...
     int myLocalVar = 0; // Local variable do NOT get a default value!
     // ...
     myLocalVar = myLocalVar + 8;
}
```

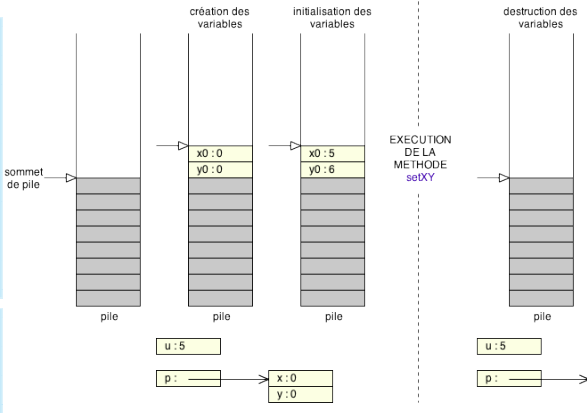
- Attribute:
  - The scope is the **entire class**.
  - An attribute is thus accessible in **all methods** of the class.
- Method parameter:
  - The scope is the **entire method**.
  - The parameter is **created at the beginning** of the method's execution.
  - It **disappears at the end** of the method's execution.

Parameter passing: How does it work?

- Java uses **pass-by-value** for parameter passing.
  - This is implemented using a **stack**.

```
public class Point {
    private int x;
    private int y;
    public void setXY(int x0,
                      int y0) {
        this.x = x0;
        this.y = y0;
    }
}
```

```
Point p = new Point();
int u = 5;
p.setXY(u, u + 1);
```



Strings: The `String` class

- String: **a sequence of characters**.
  - For example, the word `Java`.
- In Java (and almost all languages like C), a string is written between double quotes (`" "`). For example:

```
"Java"
"Java is easy"
```

- Strings are objects of the `String` class, which is provided by the JDK.

```
String str1 = null;   // Initialized with the null value
String str2 = "Java"; // Initialized with a reference
```

- On the web, search "JAVA SE String", to explore all its available methods.

String concatenation

- The `+` operator can take two strings as parameters.

```
String str1 = "Java";
String str2 = "coffee";
String str3 = str1 + " " + str2;
```

- The `+` operator concatenates its arguments:

```
String str4 = str1 + str2;
```

- This is equivalent to calling the `concat()` method:

```
String str4 = str1.concat(str2);
```

Equality operator review

- We have seen the equality test operator `x == y`.
- It only works with **discrete** scalar types:
  - `char`, `byte`, `short`, `int`, `long`, `boolean`.
  - It does not work with `float` and `double`.
- What about reference types?
  - The `==` operator, when used with a reference type, checks if both operands refer to **the same object in memory.**
- To test if two objects are equal in a broader sense, classes offer a method named `equals()`.
  - For example, to test the equality of two strings `str1` and `str2` (to check if they contain the same characters in the same order), we use

```
str1.equals(str2);
```

The `ArrayList<E>` class

- If `E` is a class (for example, `Robot`), an object of the `ArrayList<E>` class is a list of references to objects of the `E` class.

```
ArrayList<Robot> myRobots = new ArrayList<Robot>();
```

- This list is empty when instantiated (it contains no elements), but it will grow automatically as elements are added.
- The elements in the list are indexed starting from `0`.

The `ArrayList<E>` class

- The content of the list is modified by the following methods:
  - `boolean add(E eObject)`
  - `boolean add(int index, E eObject)`
  - `boolean remove(int index)`
  - `... etc.`
- Be careful with errors when using these methods:
  - Use the `size()` method to check if there is an element at the specified index.

```
Robot myRobot;
if (index > -1 && index < myRobots.size() {
    myRobot = myRobots.get(index);
}
else {
    myRobot = null;
}
```

Generic classes

- The `ArrayList<E>` class is what we call a **generic** class.
- It is a class that is **parametrized** by another class.
  - The `ArrayList<E>` class is parametrized by the class `E`.

```
ArrayList<Robot>
```

- The **programming** of generic classes is not covered in the syllabus.
- Only the **use** of predefined generic classes is covered.

`for` loop: Standard syntax

```
ArrayList<Robot> myRobots = new ArrayList<Robot>();
myRobots.add(/*... */);
// ...

for (int index = 0; index < myRobots.size(); index++) {
    Robot myRobot = myRobots.get(index);
    System.out.println(myRobot);
    // ...
}
```

- Just like in C, the `index` variable will take the values from `0` to the size of the list `-1`.
- The `myRobot` variable will take the values of all the elements in the `myRobots` structure.

`for` loop: Simplified syntax

```
ArrayList<Robot> myRobots = new ArrayList<Robot>();
myRobots.add(/*... */);
// ...

for (Robot myRobot : myRobots) {
    System.out.println(myRobot);
    // ...
}
```

- The variable `myRobot` will successively take on the value of each element in the `myRobots` structure.
  - References to objects of type `Robot`.
- This traversal method can be used with all data structures provided by the JDK, starting from version 5.

Method programming

- To complete the project, you will need to learn more about the imperative programming constructs of Java.
  - There are several other control statements, such as `if`, `while`, `switch`, etc.
  - Fortunately, this syntax is similar to that of C, which you already know.
- Given the limited time available for this course, you will need to learn this syntax by yourself.
- For this, you can download and read a course overview here.
- Also, many learning resources are available online, such as the following site: https://www.w3schools.com/java/default.asp.