# Chapter 2. Encapsulation

## 1. Concept of Encapsulation

a. Encapsulation refers to the fact that an object contains its own attributes and methods. The details of an object's implementation are hidden from other objects in the object-oriented system. This is referred to as the encapsulation of an object's data and behavior.

b. Encapsulation is a mechanism that involves grouping data and methods within a structure while hiding the object's implementation. This means preventing access to data through any means other than the provided services. Encapsulation thus ensures the integrity of the data contained within the object. The following Example 1 illustrates the concept of encapsulation, which is expressed through the visibility modifiers **private**, **public**, and **protected**.

```
public class Rectangle {
    private int width;
    protected int length;

    // Constructor to initialize the attributes
    public Rectangle(int x, int y) {
        this.width = x;
        this.length = y;   }

    // Method to display the values of width and length
    public void display() {
        System.out.println("width: " + width + " length: " + length);}}

public class Test_Rect {
    public static void main(String[] args) {
        // Creating an object of Rectangle class
        Rectangle r = new Rectangle(45, 15);
        // Calling the display method to show values
        r.display(); // corrected method name    }}
```

**Example 1**: In this example, encapsulation is interpreted by the words:
**private, public, protected**

### 2.1. Advantages of Encapsulation:

#### 2.1.1. Preservation of Object Integrity:

Encapsulation allows assigning a visibility level (access authorization for **classes**, **interfaces** (methods), or **attributes** of any given class) using the modifiers **private**, **public**, **protected**, or **no modifier**. This ensures that:

a) Attributes cannot **accept** arbitrary values.
b) Access to classes and interfaces (methods) **is restricted**.

### 2.1.2. Exception Management:

Encapsulation also provides for handling any attempt, during program execution, to violate the integrity of an object by assigning unacceptable values to its attributes. Such an attempt can be managed using an exception-handling mechanism. This mechanism allows either the class itself or the interacting class to anticipate and handle the response to this failed attempt. Possible responses include: interrupting the program, the interacting class trying another value, or continuing as if nothing happened, but without making the change.

Exception handling is a sophisticated programming mechanism designed to create more robust code. It allows anticipating and managing problems that may arise during code execution in contexts where the programmer does not have complete control. Exception handling could easily be the subject of an entire chapter on its own.

In summary**, the advantages of encapsulation are:**

- ✓ Simplification of object usage,
- ✓ Improved program robustness,
- ✓ Simplified overall application maintenance.

## 2. Visibility Levels

Encapsulation involves exposing only what is necessary for using an object while hiding other details. It enables providing the users of a class with a list of methods and, if applicable, attributes that can be accessed externally.

This list of exportable services is referred to as the class interface, consisting of a set of methods and attributes designated as **public** (public).

Methods and attributes reserved for implementing the internal behaviors of an object are called **private** (private). Their usage is exclusively restricted to the methods defined within the current class.

The table below demonstrates the visibility levels **(access rights**) that can be assigned to attributes, methods, and classes:

| | **public** | **protected** | **default** | **private** |
|---|---|---|---|---|
| **in the same class** | yes | yes | yes | yes |
| **in a class of the same package** | yes | yes | yes | no |
| **in a subclass of another package** | yes | yes | no | no |
| **in any class of another package** | yes | no | no | no |

Table 1. access authorization

### 2.1. Explanation:

**For the first column of the table** (**public**):This means that a class or a class member (attribute or method) with the **public** modifier is accessible everywhere. It can be accessed in the same class, in a class within the same package, in a subclass (will discuss in the next chapter) of another package, and in any other class in a different package.

**For the last column of the table (private)**: This means that a class member with the **private** modifier is only accessible within the class where it is declared.

### 2.3. Summary Tables of Access Rights

#### a. Access Modifiers for Classes and Interfaces

| Modifier | Meaning for a Class or Interface |
|---|---|
| public | Always accessible |
| None (no modifier) | Accessible only from classes within the same package |

#### b. Access Modifiers for Members and Inner Classes

| Modifier | Meaning for Members and Inner Classes |
|---|---|
| public | Accessible anywhere the class is accessible |
| None (no modifier) | Accessible from all classes within the same package |
| protected | Accessible from all classes in the same package or derived classes |
| private | Access restricted to the class where the member or inner class is declared |

### 2.3.4 Concept of a Package:

A package is a library that groups a large number of classes provided by Java SE. These classes implement generic data and processes that can be used by many applications. These classes constitute the API (Application Programmer Interface) of the Java language. An online documentation for the Java API is available at:

http://docs.oracle.com/javase/7/docs/api/

All these classes are organized into packages (or libraries) dedicated to specific themes. Some of the most commonly used packages include the following:

| Package | Description |
|---|---|
| java.awt | Graphic classes and interface management |
| java.io | Input/output management |
| java.lang | Core classes (imported by default) |
| java.util | Utility classes |
| javax.swing | Additional graphic classes |

To access a class from a given package, you must first import that class or its package. For example, the **Date** class belonging to the **java.util** package, which implements a set of methods for date manipulation, can be imported in two ways:

A single class from the package is imported: **import java.util.Date;**

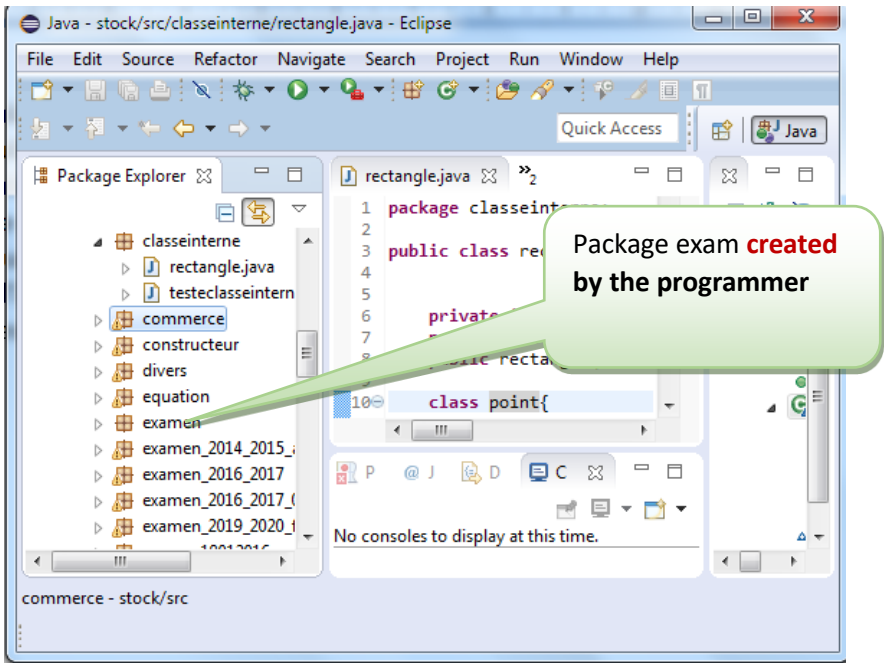
All classes from the package are imported (including unused ones): **import java.util.*;**

The following program uses this class to display the current date:

```
import java.util.Date;
public class DateMain{
public static void main(String[]args){
Datetoday= newDate();
System.out.println(”Noussommes le”+today.toString());}}
```

### 2.3.4.1 Package with an IDE: Eclipse

The following image shows an example of a package created by a programmer.




### a. Data (attributes) and method (message) encapsulation

Encapsulation allows defining visibility levels for the elements of a class. These visibility levels define the access rights to data depending on whether it is accessed by a method of the class itself, a subclass (**see the next chapter**), or any other class. There are three visibility levels:

- **public:** functions from all classes can access the data or methods of a class defined with the public visibility level. **This is the lowest level of data protection.**
- **protected:** access to data is reserved for functions of the subclass, meaning by the member functions of the class as well as derived classes.
- **private:** access to data is restricted to the methods of the class itself. **This is the highest level of data protection.**

**Example 2**: Let's revisit **Example 1** of the Rectangle class.

• If length and width were public attributes of Rectangle, the following code could be written in the "main" method of the Test_Rect class:

```
Rectangle r = new Rectangle();
r.length = 20;
r.width = 15;
int la = r.length;
```

If length and width were private attributes, the following instructions would be rejected by the compiler:

```
r.length = 20; // incorrect
r.width = 15; // incorrect
int la = r.length; // incorrect
```

In the second case, you would need to define access methods setLength(int) and setWidth(int) to modify the attribute values, and access methods getLength() and getWidth() to return the length and width values of a Rectangle object. In this case, the instructions would be as follows:

```
r.setLength(20); // correct
r.setWidth(15); // correct
int la = r.getLength(); // correct
```

### 3. Encapsulation in Java

#### a. Access Control (public, private)

In the previous examples, the keyword public sometimes appears at the beginning of a class or method declaration, which allows any object to use the class or method declared as public. The scope of this authorization depends on the element it applies to. Table 2 presents the scope of these authorizations.

| Element | Authorization |
|---|---|
| Variable | reading and writing |
| Method | method call |
| Class | Object instantiation of this class and access to class variables and methods |

Table 2. the scope of authorization

The **public** mode is, of course, not the only type of access available in Java. Three other keywords can be used in addition to access type: the **default**, **protected** and **private**. Table 3 summarizes these different access types (**the concept of subclass is explained in the next chapter**).

**Example 3:**

Consider the following program, which consists of two packages, p1 and p2. Package p1 contains three classes, C1, C2, and C3, where C2 is a subclass of C1 (**see inheritance in the next chapter**). Package p2 contains two classes, C4 and C5, where C4 is a subclass of C1.

```
package p1 ;
class C1 {
public int a;
protected int b;
int c;
private int d;}
class C2 extends C1 {}
class C3 {}
```

```
package p2 ;
import C1.p1;
class C4 extends C1 {}
class C5 {}
```

Table 4 presents the accessibility for the variables a, b, c, d by the classes C1, C2, C3, C4, C5.

| | Variable (a) | Variable (b) | Variable (c) | Variable (d) |
|---|---|---|---|---|
| the accessibility of C2 | oui | oui | oui | non |
| the accessibility of C3 | oui | oui | oui | non |
| the accessibility of C4 | oui | oui | non | non |
| the accessibility of C5 | oui | non | non | non |

Table 4: Accessibility of variables by class according to the program in Example 3.

**b. Accessors (get and set)**

To maximize the security of a class's attributes, the **private** visibility modifier is always used, which makes the attribute inaccessible (whether for reading or writing) outside the class where it is declared. This requires the implementation of accessors for reading and writing, commonly known as **setters and getters**.

**Example 4:** Let's revisit Example 1 of the Rectangle class with a small modification.

```
public class Rectangle {
   private int length;
   private int width;

   // First constructor
   Rectangle(int l, int a) {
      length = l;
      width = a; }
```

```
    // Second constructor
    Rectangle() {
       length = 20;
       width = 10; }

    // Third constructor
    Rectangle(int x) {
       length = 2 * x;
       width = x;  }

    // Get method for length
    public int getLength() { return (length); }

    // Get method for width
    public int getWidth() { return (width); }

    // Set method for length
    public void setLength(int l) {  length = l; }

    // Set method for width
    public void setWidth(int l) { width = l; }

    // Method to calculate area
    public int surface() { return (length * width);}

    // Method to calculate perimeter
    public int perimeter() { return ((width + length) * 2); }

    // Method to extend the length of a rectangle
    public void extendLength(int l) { length += l;  }

    // Method to display rectangle's characteristics
    public void display() {System.out.println("Length=" + length + " Width=" + width);
    }
}
```

**Code for the Test_Rect class:**

```
class Test_Rect {
    public static void main(String[] args) {
        // Creating objects of Rectangle with different constructors
        Rectangle r = new Rectangle(10, 5);
        Rectangle r3;
        r3 = new Rectangle(14);
        Rectangle r2 = new Rectangle();
```

```
        // Displaying characteristics of rectangles
        r.display();
        r2.display();
        r3.display();

        // Modifying the dimensions of r2
        r2.setLength(50);
        r2.setWidth(30);
        r2.display();

        // Printing the surface and perimeter of the first rectangle
        System.out.println("Rectangle 1");
        System.out.println("Surface= " + r.area());
        System.out.println("Perimeter= " + r.perimeter());

        // Extending the length of r and displaying the updated values
        r.extendLength(40);
        System.out.println("After extending:");
        r.display();
    }
}
```

**c. Access to the instance (this)**

You can use **this** as an object or **this** (parameters) as a method.

- ✓ In a constructor or an instance method, the object represented by **this** is the object you are currently working with.
- ✓ **this** is typically used to access an instance attribute that is shadowed by a method argument or a local variable.
- ✓ You can also use **this** to pass the object you are working with as a parameter to a method.
- ✓ Invoking a method **this** (parameters) can only be done in the first line of a constructor. In this case, another constructor is invoked whose parameter list matches the types of the parameters in the parentheses.

**Example 5:**

```
class Circle {
    int x, y, radius;
    Circle(int x, int y, int radius) {
        this.x = x;
        this.y = y;
        this.radius = radius; // this is used as an object
    }
    Circle(int x, int y) {  this(x, y, 10); // this is used as a method  }}
```

**d. Class variables and methods (static)**

**d.1. Instance variables:**

- ✓ Are declared outside any method.
- ✓ Preserve the state of an object, an instance of the class.
- ✓ Are accessible and shared by all methods in the class.

**d.2. Static variables and methods:**

- ✓ Some variables are shared by all instances of a class. These are the class variables (**static** modifier).
- ✓ No need to create an object (instances of the class).

**d.3. Static method:**

- ✓ Provide functionality for the entire class.
- ✓ Methods that are not intended to operate on an individual object of the class.
- ✓ Examples: Math.random(), Integer.parseInt(String s), main(String args[]).
- ✓ Static methods cannot access instance variables.

**d.4. The reserved word static (called modifier):**

This modifier applies to attributes and methods; when this modifier is used, it is said to be a class attribute or method.

- ✓ It is important to remember that a method or an attribute to which the **static** modifier is not applied is called an instance method or attribute.
- ✓ A **static** attribute exists as soon as its class is referenced, independently of any instantiation. Regardless of the number of class instantiations (0, 1, or more), a class attribute (i.e., static) exists in one and only one instance. Such an attribute is used somewhat like a global variable in a non-object program.
- ✓ A method, to be a class method (i.e., **static**), must not manipulate, either directly or indirectly, non-static attributes of its class. As a result, a class method cannot directly use any non-static attribute or method of its class; an error would be detected at compilation. In other words, a method that uses (for reading or writing) instance attributes cannot be **static** and must, therefore, be an instance method.

From outside a class or a derived class (see the next chapter), a class attribute or method can be used preceded by the name of the class:

**class_name.class_data_or_method_name**

Or, though less correct, by the name of an instance of the class.
Finally, note that a **static** method cannot be overridden (see the next chapter), which means that it is automatically **final**.

```
public class MyClass {    static int counter = 0;}
```

The ownership of class variables by an entire class and not by a specific object allows replacing the variable name with the class name.

Continuation of Example 6:

```
MyClass m = new MyClass();
int c1 = m.counter;
int c2 = MyClass.counter;
```

c1 and c2 have the same value.

This type of variable is useful, for example, to count the number of instantiations of the class that have been made.

**Chapter end**