Lesson 4.2

Encapsulation

By the end of this lesson you will learn:

- Introduction to Encapsulation
- Why is Encapsulation
- Information Hiding in Java
- Access Modifiers
- Accessibility matrix

Introduction to Encapsulation

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It is the mechanism that binds together code and the data it manipulates and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. It describes the idea of bundling data and methods that work on that data within one unit, e.g., a class in Java.

To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever. You, as the user, have only one method of affecting this complex encapsulation: by moving the gear shift lever. You cannot affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a well-defined (indeed, unique) interface to the transmission. Further, what occurs inside the transmission does not affect objects outside the transmission. For example, shifting gears does not turn on the headlights! Because an automatic transmission is encapsulated, dozens of car manufacturers can implement one in any way they please. However, from the driver's point of view, they all work the same. This same idea can be applied to programming. The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.

This concept is also often used to **hide the internal representation**, or state, of an object from the outside. This is called **information hiding**. The general idea of this mechanism is simple. If you have an attribute that is not visible from the outside of an object, and bundle it with methods that provide read or write access to it, then you can hide specific information and control access to the internal state of the object.

Why is Encapsulation

It is because encapsulation has a number of advantages that increase the reusability, flexibility and maintainability of the code.

- **Flexibility**: It's more flexible and easier to change the encapsulated code with new requirements.
- **Reusability**: Encapsulated code can be reused throughout the application or across multiple applications.
- **Maintainability**: Application ode is encapsulated in separate units, so it's easy to change or update a part of the application without affecting other parts, which reduces the time of maintenance.

Information Hiding in Java

As explained at the beginning, we can use the encapsulation concept to implement an information-hiding mechanism. Like the abstraction concept, this is one of the most used mechanisms in Java. We can find examples of it in almost all well-implemented Java classes.

We implement this information-hiding mechanism by making our class attributes inaccessible from the outside and by providing getter and/or setter methods for attributes that shall be readable or updatable by other classes.

See the below example:

```
class Person {
    private String name;
    private int age;

    public void setName(String name) {
       this.name = name;
    }

    public void setAge(int age) {
       this.age = age;
    }

    public String getName() {
        return name;
    }

    public String getAge() {
        return age;
    }
}
```

Here, the fields **age** and **name** can be only changed within the Person class. If someone

attempts to make a change like this:

```
Person p = new Person();
p.name = "Tom"; // ERROR!
```

The code will not compile because the field **name** is marked as **private**.

Access Modifiers

The three access modifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories.

- **public:** Anything declared **public** can be accessed from anywhere.
- **private:** Anything declared **private** cannot be seen outside of its class.
- **protected:** If we want to allow an element to be seen outside our current package, but only to classes that subclass your class directly, then declare that element **protected**.
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the **default access**.

Accessibility matrix

Here you can see an overview of the different access modifiers and the accessibility of the attributes or methods.

Access Modifier	Within Class	Within Package or Folder	Outside Package by Subclass only	Outside Package or Folder
Public	Yes	Yes	Yes	Yes
Private	Yes	No	No	No
Protected	Yes	Yes	Yes	No
Package/Default	Yes	Yes	No	No

Exercise

1. Write a Java program to create the following class.