**Module 3**

**Encapsulation on Object-Oriented Programming**

**SAFRIZAL RAHMAN\_19\_SIB\_2G**

**Link :** [**https://github.com/safrizalrahman46/Week3\_Enskapsulasi.git**](https://github.com/safrizalrahman46/Week3_Enskapsulasi.git)

# 1. Purpose

After conducting experiments on this module, students understand the concept:

1. Encapsulation (access level modifiers, setters and getters)
2. Constructor
3. Understanding the notation related to access level modifiers in UML Class Diagrams

# 2. Introduction

In the first and second meetings, the basic concepts of object-based programming (PBO), the difference between object-based programming and structural programming, and the concepts of classes and objects were discussed. Furthermore, in this module, the concept of encapsulation and notation in the UML Class diagram will be discussed.

## 2.1. Encapsulation

Definition:

* Unification/merging of attributes and methods of an object into a whole
* Restrict direct access to components of an object Purpose of encapsulation:
* Concealment of the internal structure of an information → hiding/data hiding object
* Protects attributes from random changes outside of the class. Attributes can be made *read-only* or *write-only*
* Simplify the implementation of changes to requirements
* Makes system unit testing easier

Encapsulation mechanism:

* Set *the access level modifier* to private so that it cannot be accessed directly from outside the class
* Provides *getters* and *setters* as a way to access or modify private attributes

### 2.2.1. Access Level Modifier

There are 4 access level modifiers, namely:

* *public* – can be accessed from anywhere
* *protected* – can be accessed outside the package using a subclass (creating an inheritance)
* No modifier (*package-private*) – can only be accessed within the same package
* *Private* – can only be accessed within the same class

Attributes and methods have 4 types of *access level modifiers* above, but classes only have 2 types of *access level modifiers* , namely *public* and *no modifiers*.

## Table 1. 1 Access Level Modifier



### 2.2.2. Getters and Setters

Getter

* Public method that returns the value of the private attribute
* There is a return value Setter
* Public methods that function to manipulate the value of private attributes
* No return value

### 2.2.3. Read-Only and Write-Only

Read-only attribute

* Attributes that only have getters, but don't have setters
* Attribute values can be accessed from inside or outside the class • Modifying attribute values can only be done in the class.

Write-only attribute

* Attributes that only have setters, but don't have getters
* Modifying attribute values can be done from inside or outside the class
* The value of the attribute can only be accessed from the class

### 2.3. Constructor

Constructor is a method used to instantiate objects from a class. If not explicitly created, java has provided a default constructor with no parameters, meaning that the object is created without assigning an attribute value. If there is a need that requires some or attribute values to be valued when the object is created, then we need to define our own constructors.

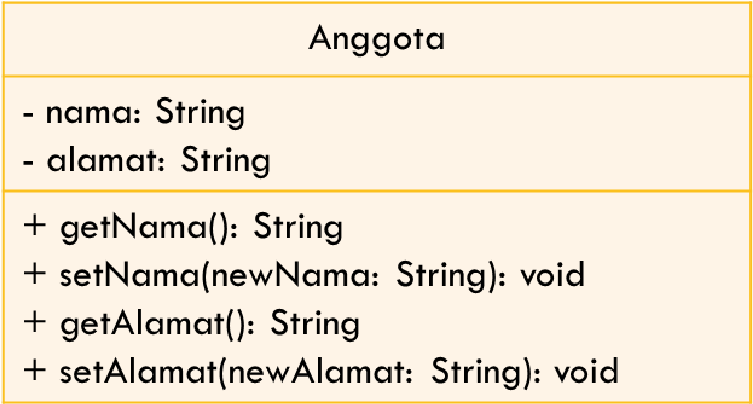
Some constructor declaration rules:

* The constructor name must be the same as the class name
* Constructors don't have a return type

### 2.4. UML Class Diagram Notation

The notation of the access level modifier in the UML class diagram is as follows:

* The plus sign (+) for public
* Hashtags (#) for protected
* Minus sign (-) for private
* For no-modifiers not given notation



**Figure 1. 1 UML Class Diagram**

# 3. Experimentation

## 3.1 Experiment 1 – No Encapsulation

In the encapsulation experiment, create a Motor class that has the attribute of the plate Number, isMachineOn (true if the engine is running and false if it is not running), and the speed and method displayStatus() to display the motor status. The UML class diagram of the Motor class is as follows:

|  |
| --- |
| Motor |
| + plateNumber: String  + isMesinOn: Boolean  + Speed: INT |
| +displayStatus(): void |

|  |
| --- |
| RakServer |
| * brand: String * capacity: int * numberOfServers: int * status: String * location: String |
| * setAddServer(server: Server): void * setRemoveServer(server: Server): void * getCheckStatus(): String * setMoveLocation(newLocation: String): void |

1. Open Netbeans or VS code, create a **Jobsheet03 project**.
2. Create a **Motor class**. Right-click on the package **jobsheet03** – New – Java Class.
3. Type the Motor class code below.



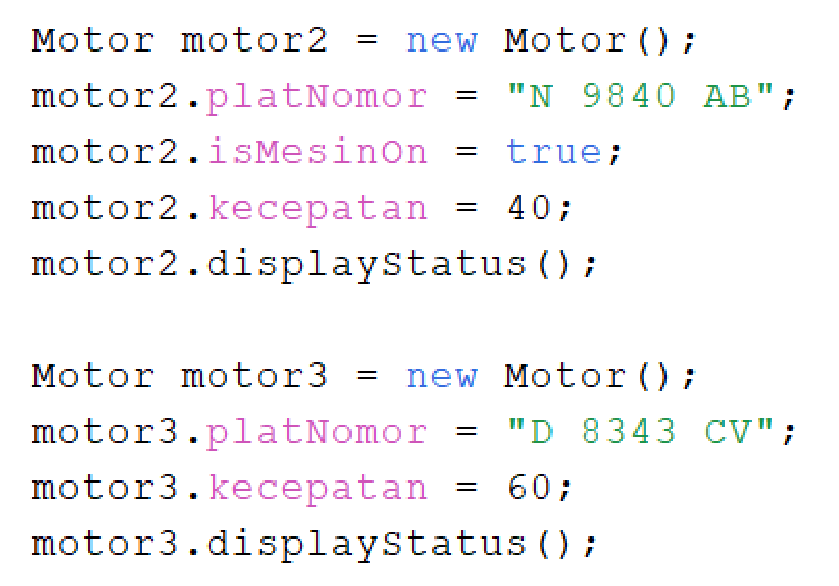
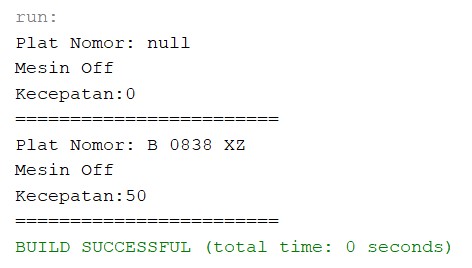
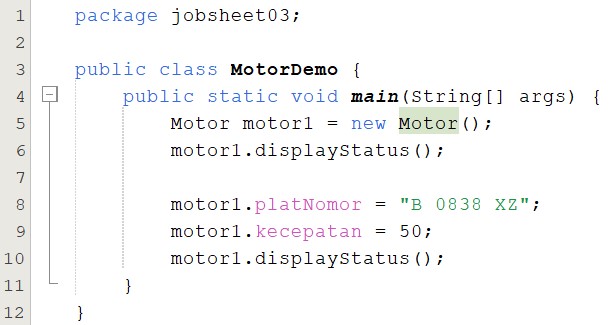
1. Then create a MotorDemo class, type the following code.

5.

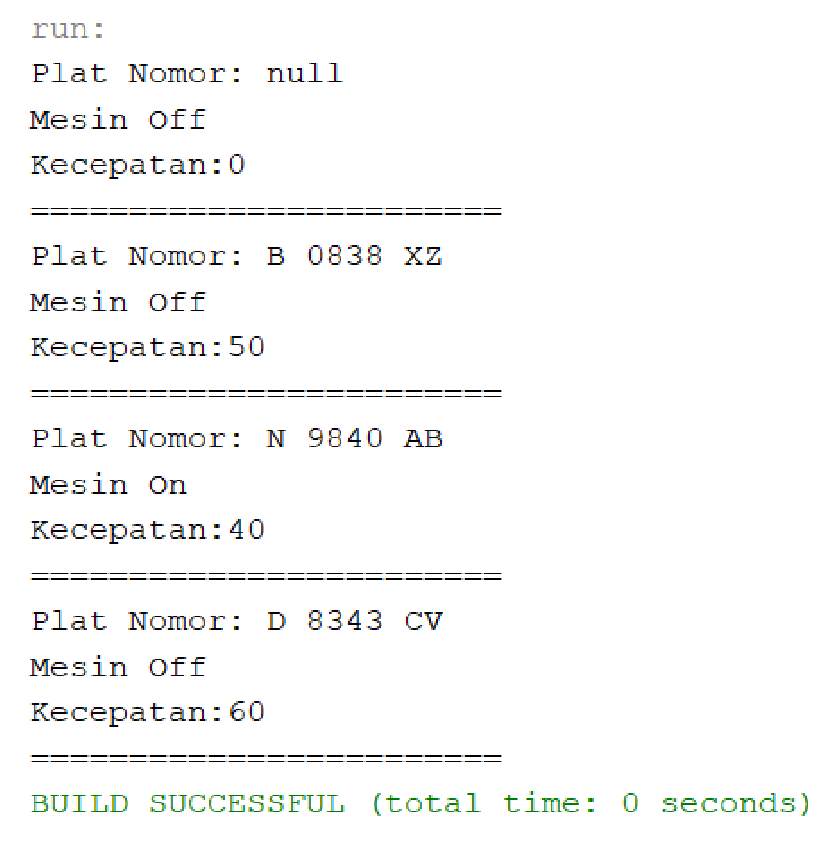
The results are as follows:

6.

Next, make 2 more motorcycle objects in class MotorDemo.java



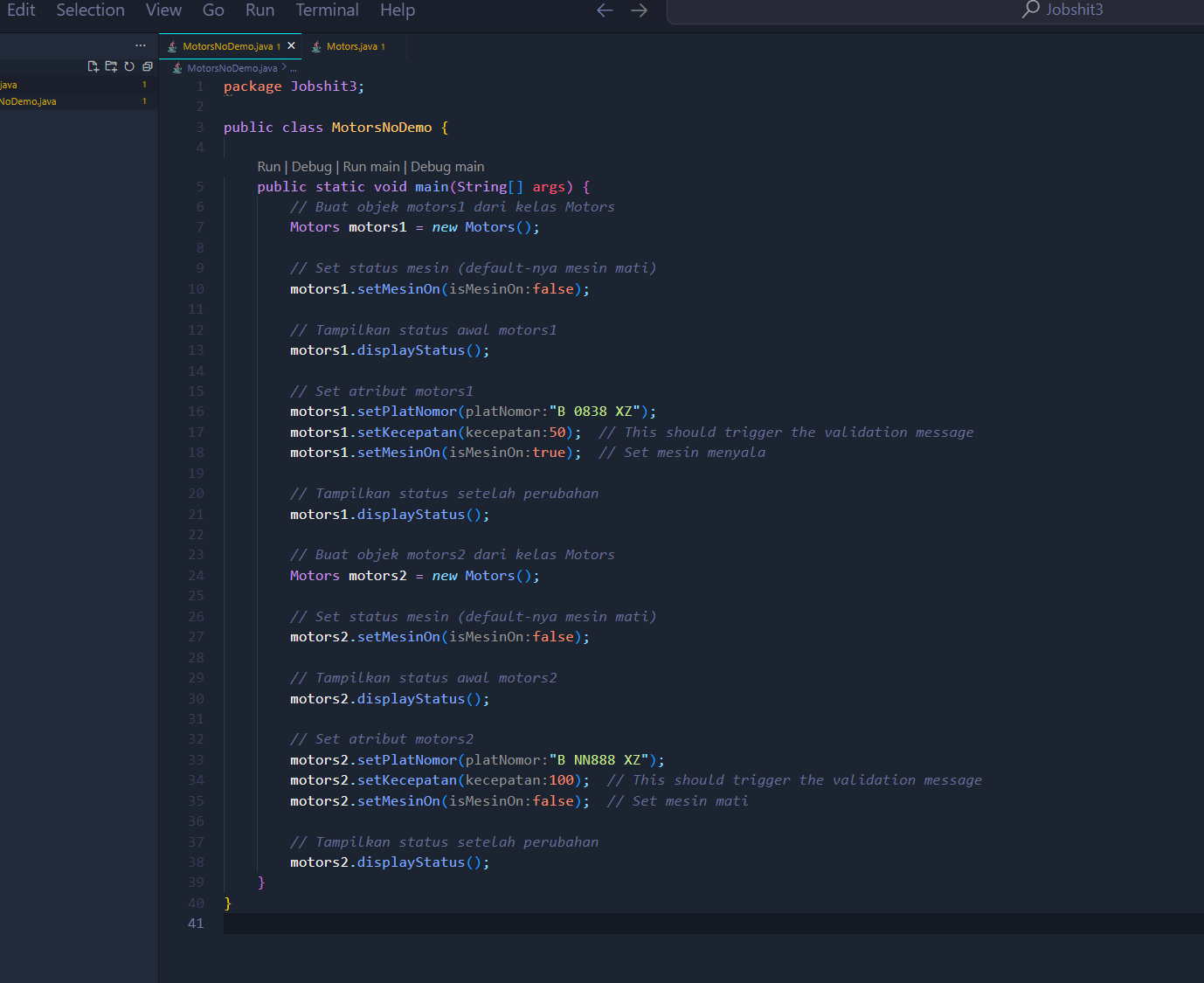
1. The results are as follows

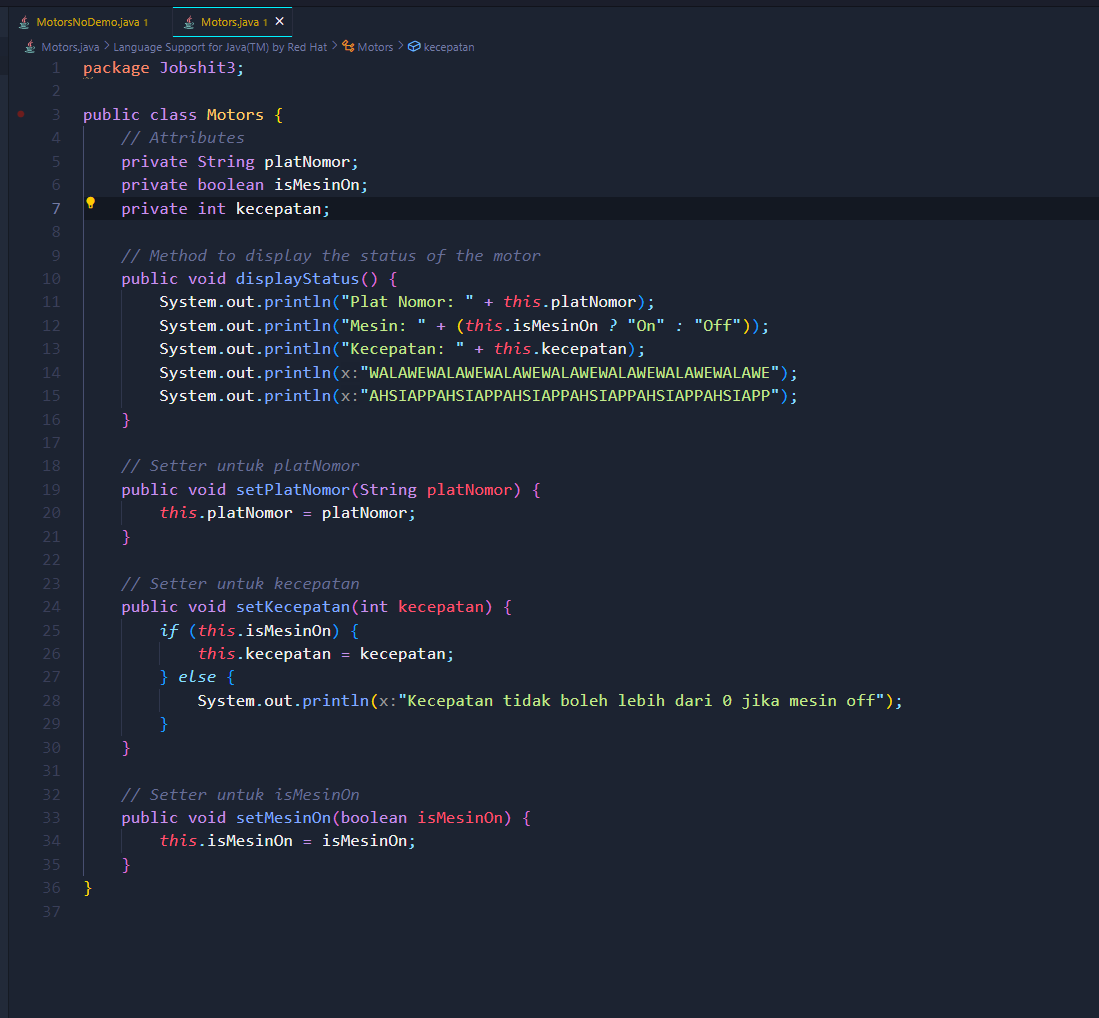


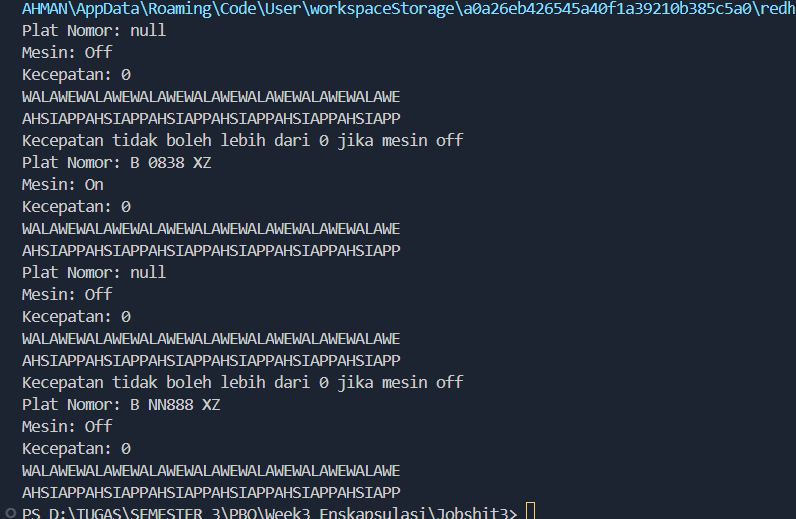


1. From the above results, is there anything strange?

On motor1 with the plate "B 0838 XZ", the speed can change from 0 to 50 even though the motorcycle engine is still Off. How is it possible for the speed attribute to be worth 50 even though the engine is still Off? This is because there is no control/restriction on speed attributes. In fact, objects in the real world always have limitations and mechanisms for how they can be used. For example, a motor that must be in a state of ignition when the speed is more than 0. This irregularity also occurred on the third motorcycle with the license plate "D 8343 CV".

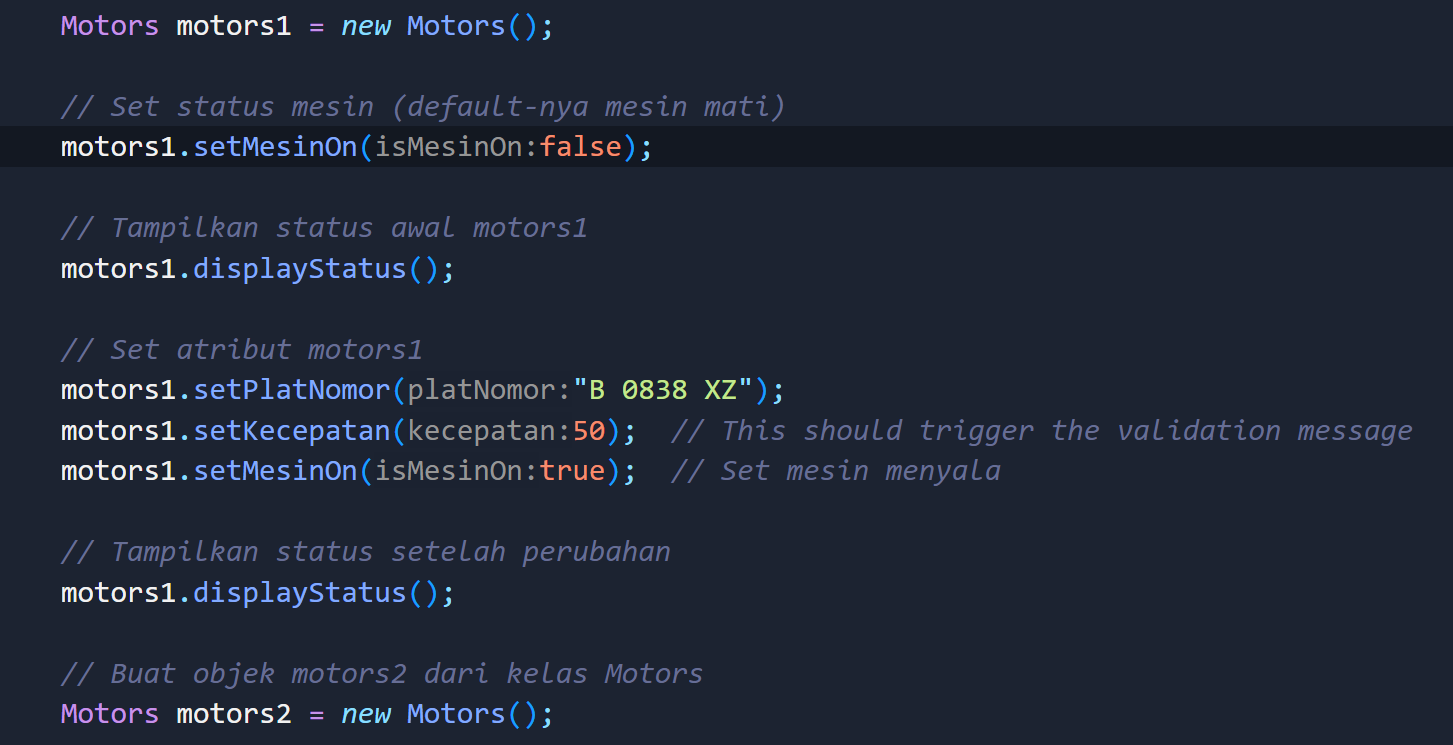






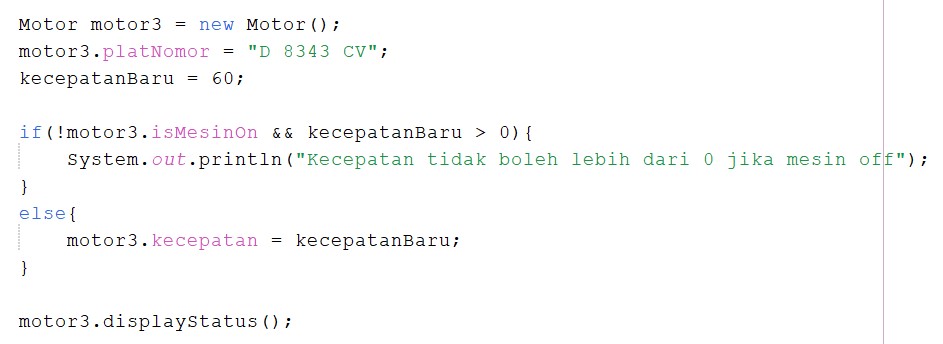
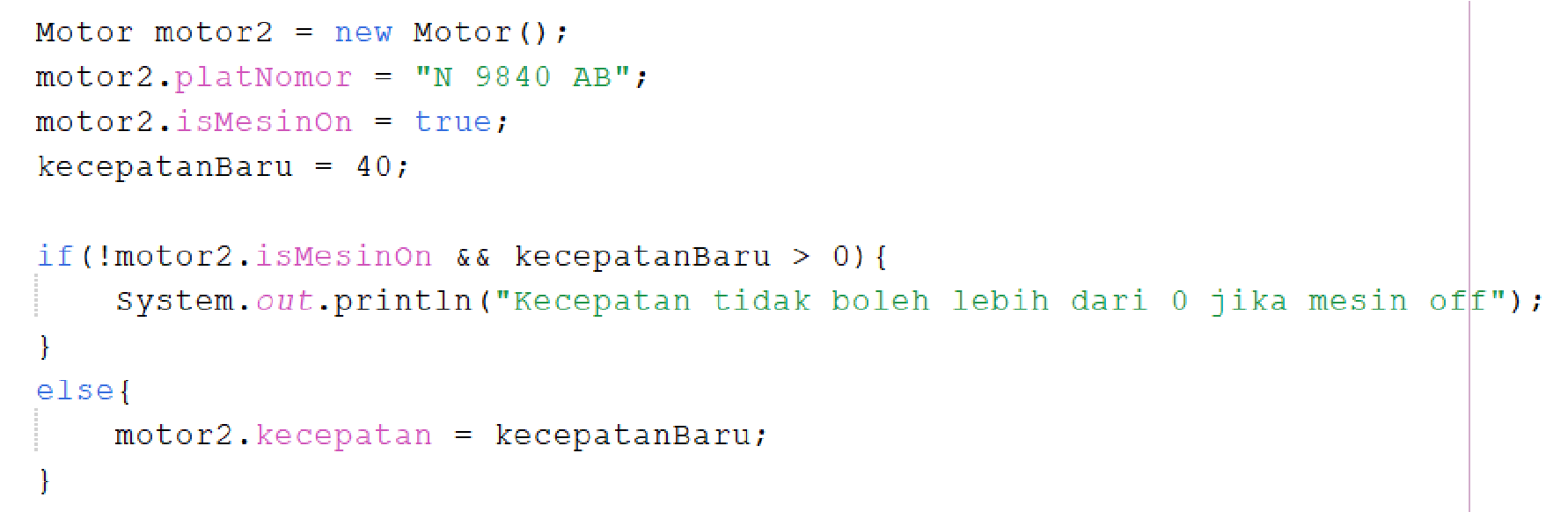
1. To overcome this, the new speed value needs to be checked first before assigning it to the speed attribute value

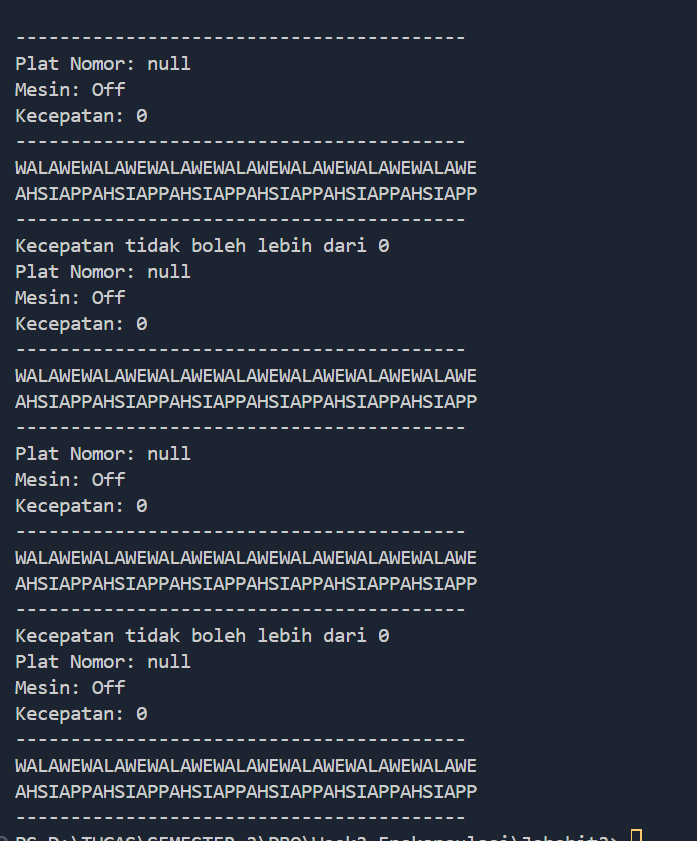




1. Perform the same check for motor2 and motor3

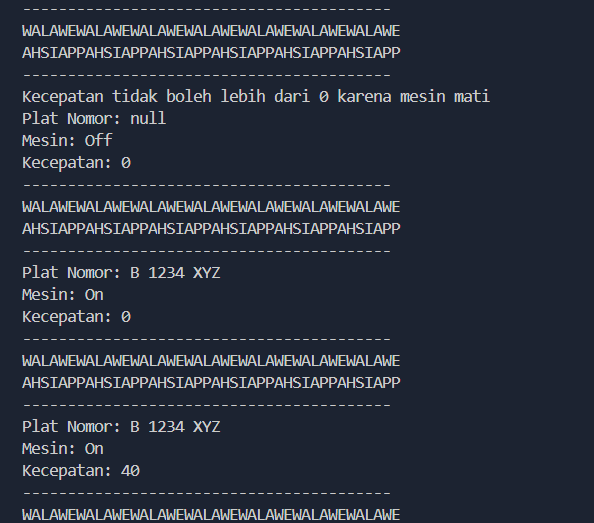






1. Run MotorDemo.java and note that there is already validation of the speed value against the engine status for each motorcycle object





## 3.2 Experiment 2 – Encapsulation

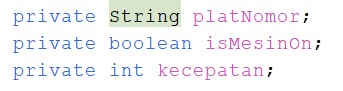
1. Imagine that the new developer remembers that the speed should not be more than 0 if the engine state does not start after creating 20 motor objects in MotorDemo.java, 10 motor objects in MotorDemo2.java, 25 objects MotorDemo3.java? Checks must be done 55 times.
2. Then, how can we improve the motorcycle class above so that it can be used properly? This is where encapsulation is important in object-oriented programming. The internal structure of the Motor class must be hidden from other classes.

In OOP, the concept of encapsulation is implemented by:

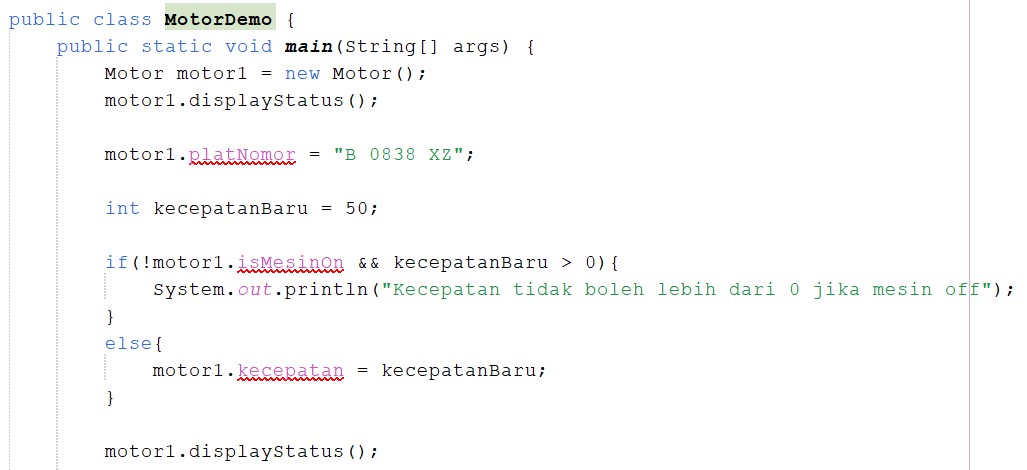
* 1. Hide internal attributes (plateNumber, isMachineOn, and speed) from other classes by changing the access level modifier to private
  2. Provides setters and getters to manipulate and access the values of those attributes

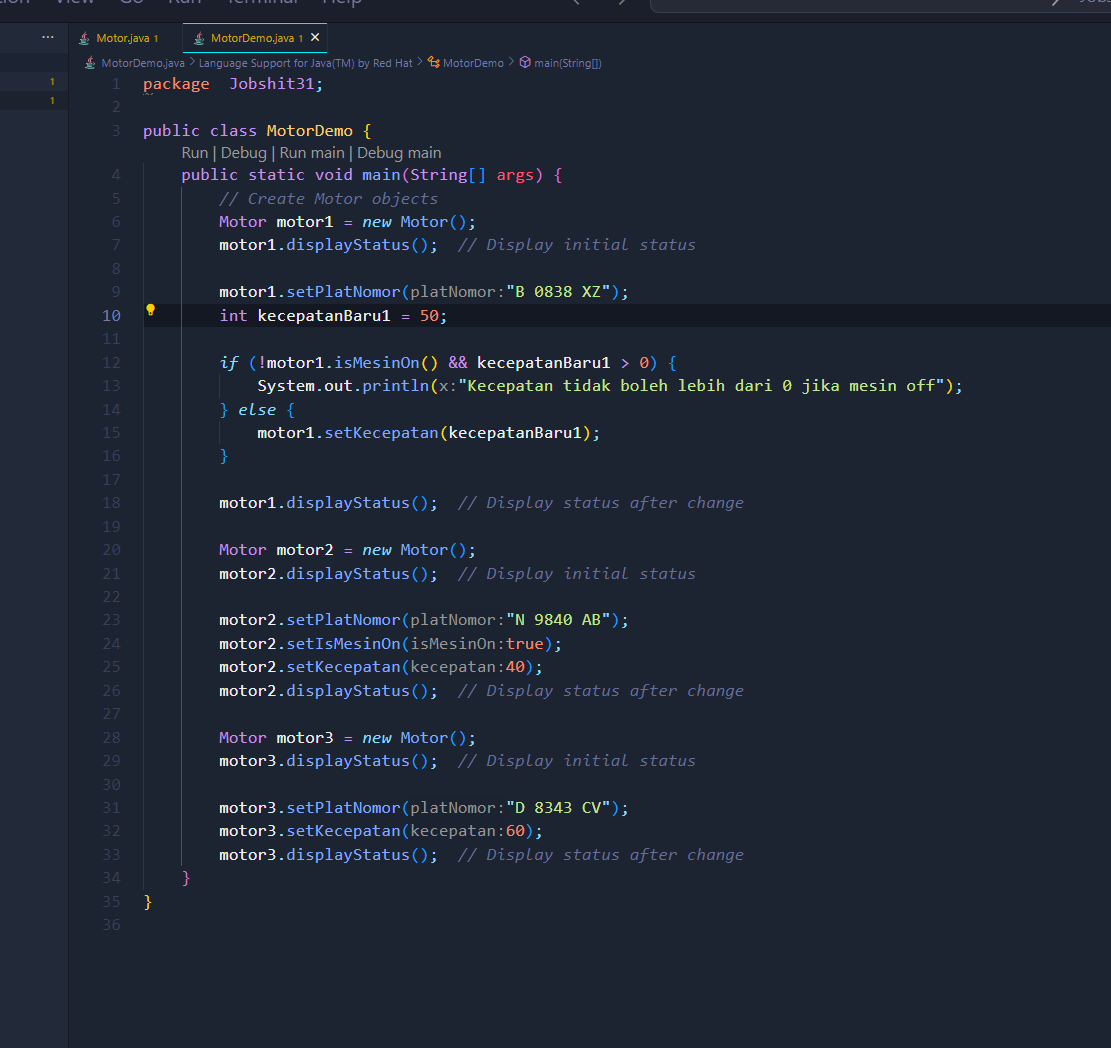
|  |
| --- |
| Motor |
| * plateNumber: String * isMesinOn: Boolean * Speed: INT |
| +displayStatus(): void  +setPlatNumber(plateNumber:String): void  +getPlatNumber(): String  +setIsMesinOn(isMesinOn:boolean): void  +getIsMesinOn(): boolean  +setSpeed(speed:int): void  +getSpeed(): int |

1. Change access level modifier to private

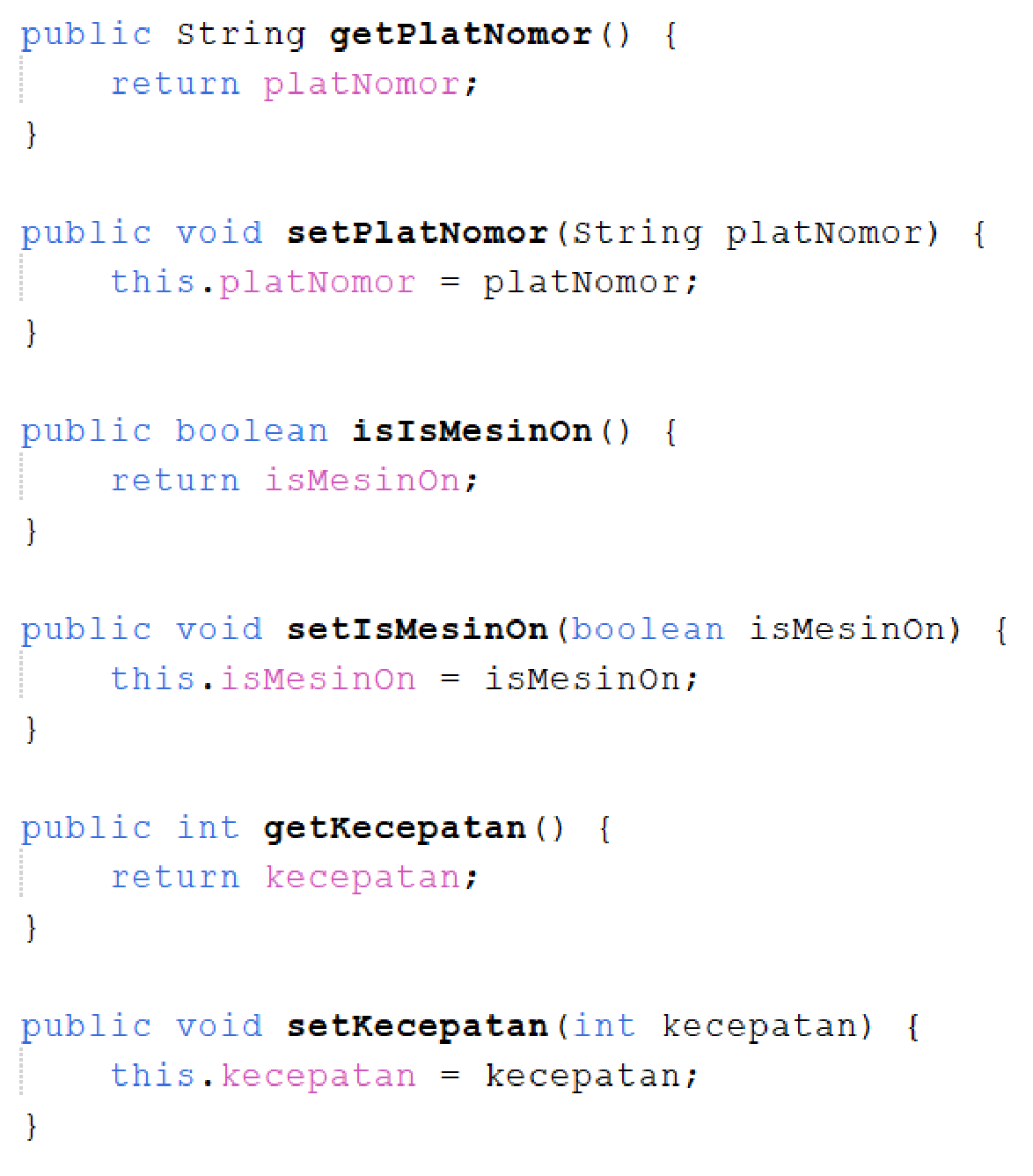


1. After changing to private, the plateNumber, isMachineOn, and speed attributes cannot be accessed from outside the class (an error appears)

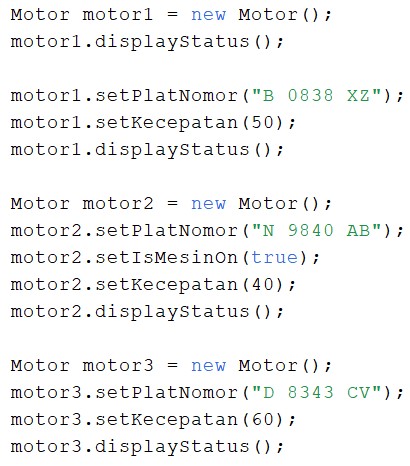


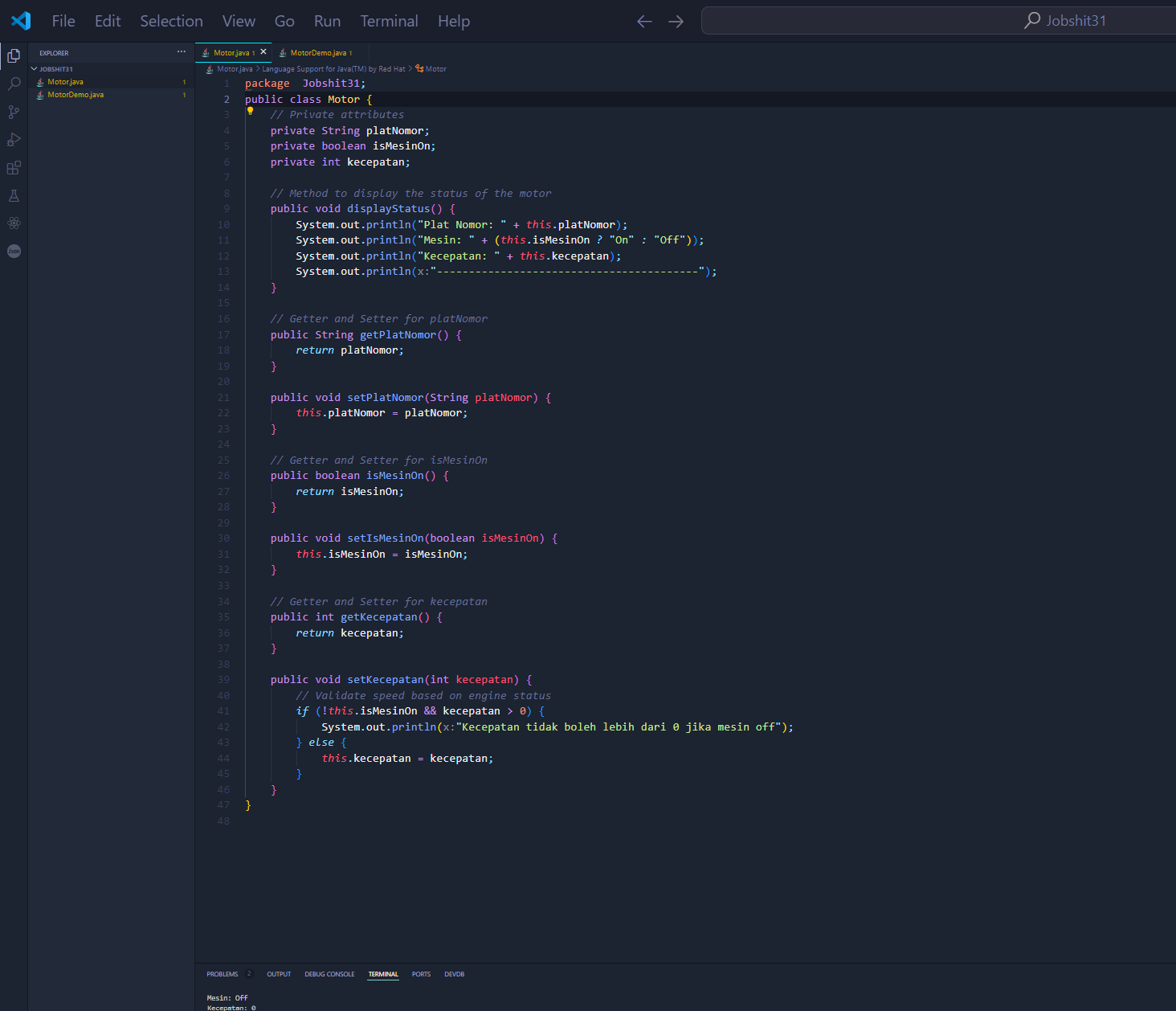


1. Next, it is necessary to create setters and getters for each attribute.

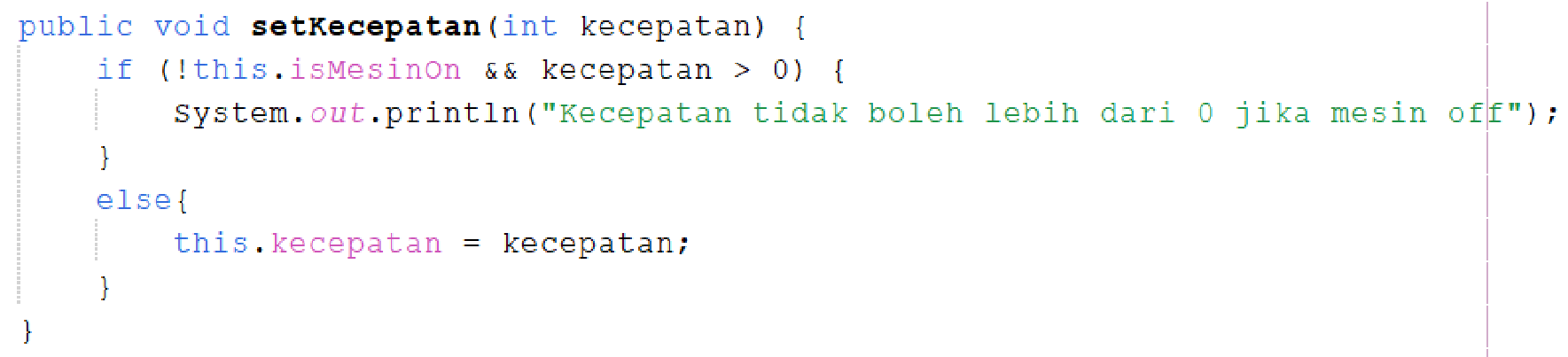


1. With encapsulation, the attribute value is accessed using getters and manipulated using the following setters (there is no validation of the speed value to the machine state yet)

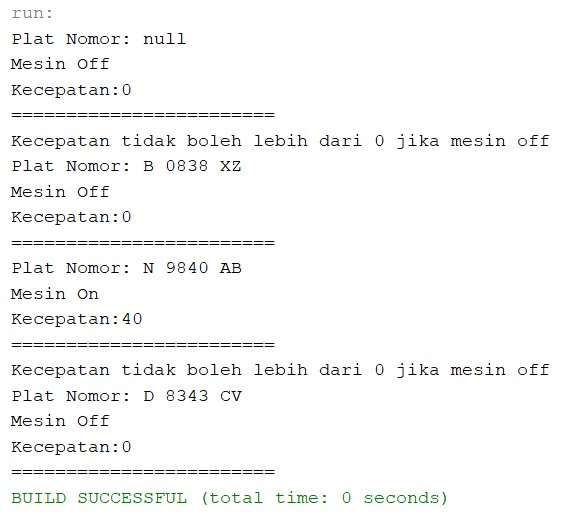


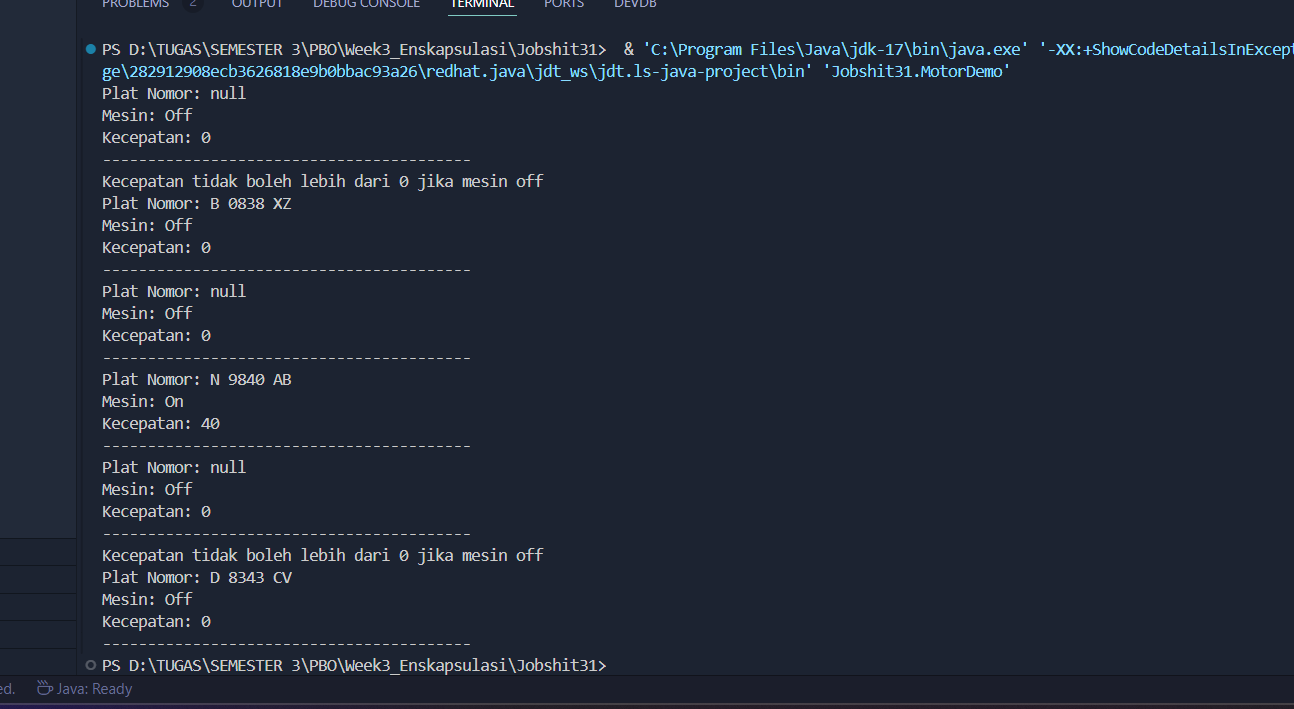


1. By implementing encapsulation, changing requirements in the midst of program implementation can be made more easily. On the speed setter, the speed value is validated against the engine status as follows:



1. MotorDemo.java run. The results are as follows:





1. Setters and getters are used as "gateways" to access or modify attributes that are of private value. This will make controlling or validating attributes easier. If there is a change in the requirement in the future, for example the speed attribute should not have a negative value, it is only necessary to make modifications to the Speed() set without the need to make repeated changes throughout the program that assigns the speed value of the motorcycle.

## 3.3 Questions

1. In the MotorDemo class, when we increase the speed for the first time, why does the warning "Speed cannot increase because the engine is off!"?

The warning appears because in your Motor class, when the isMesinOn (engine status) is false, the program restricts the speed from increasing. This is done by the logic in the setKecepatan() method. If the engine is off and the speed is greater than 0, the method prevents the speed from increasing and prints the warning message.

1. Do you want to know the brand attributes, speed, and status of the machine set private?

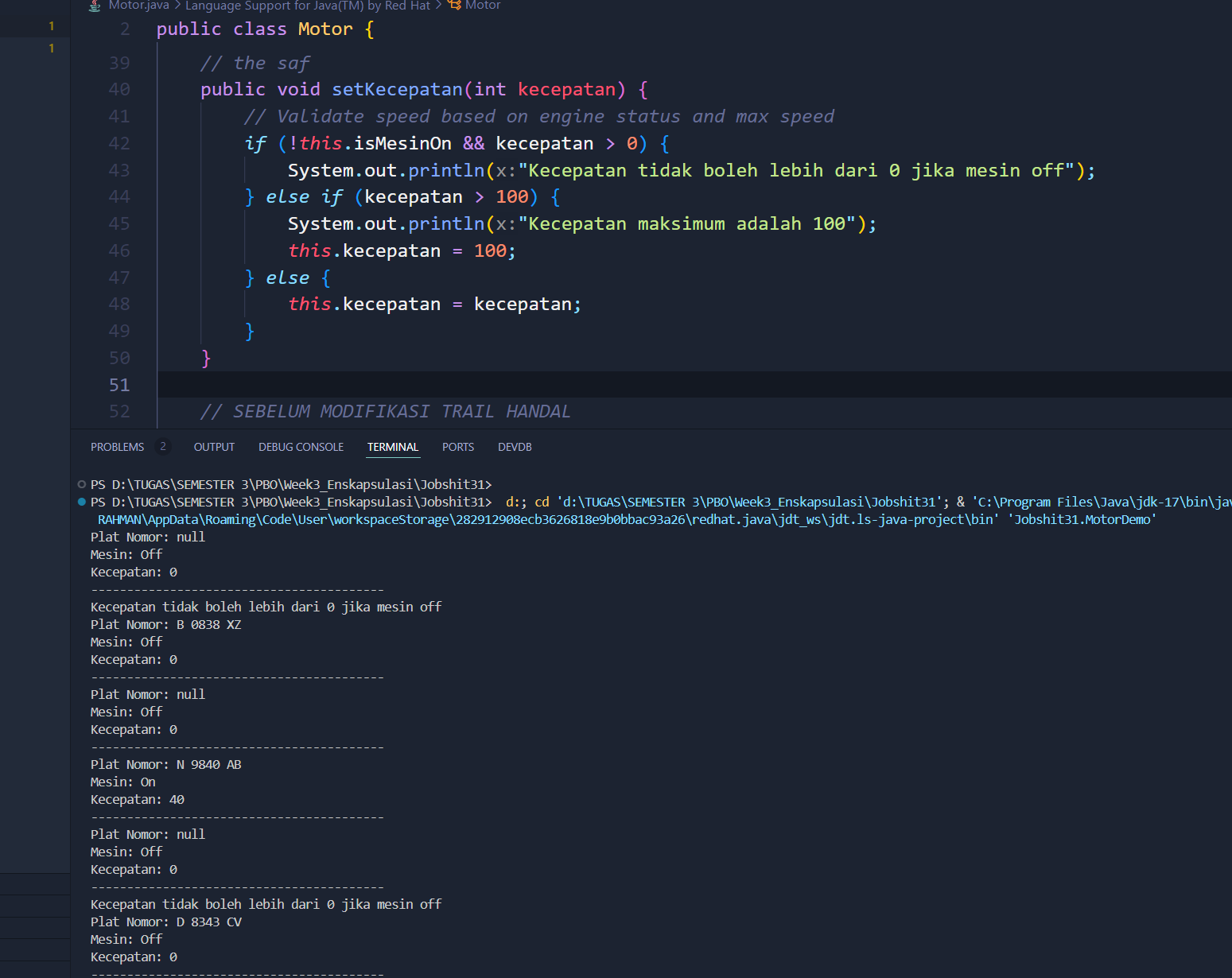
Yes, it's a good practice to keep attributes like platNomor, kecepatan, and isMesinOn private. This is part of the principle of encapsulation in object-oriented programming, which helps prevent direct modification of the attributes from outside the class and ensures better control over how data is accessed or modified. Getters and setters provide controlled access to these attributes.

1. What is the function of setter and getter?

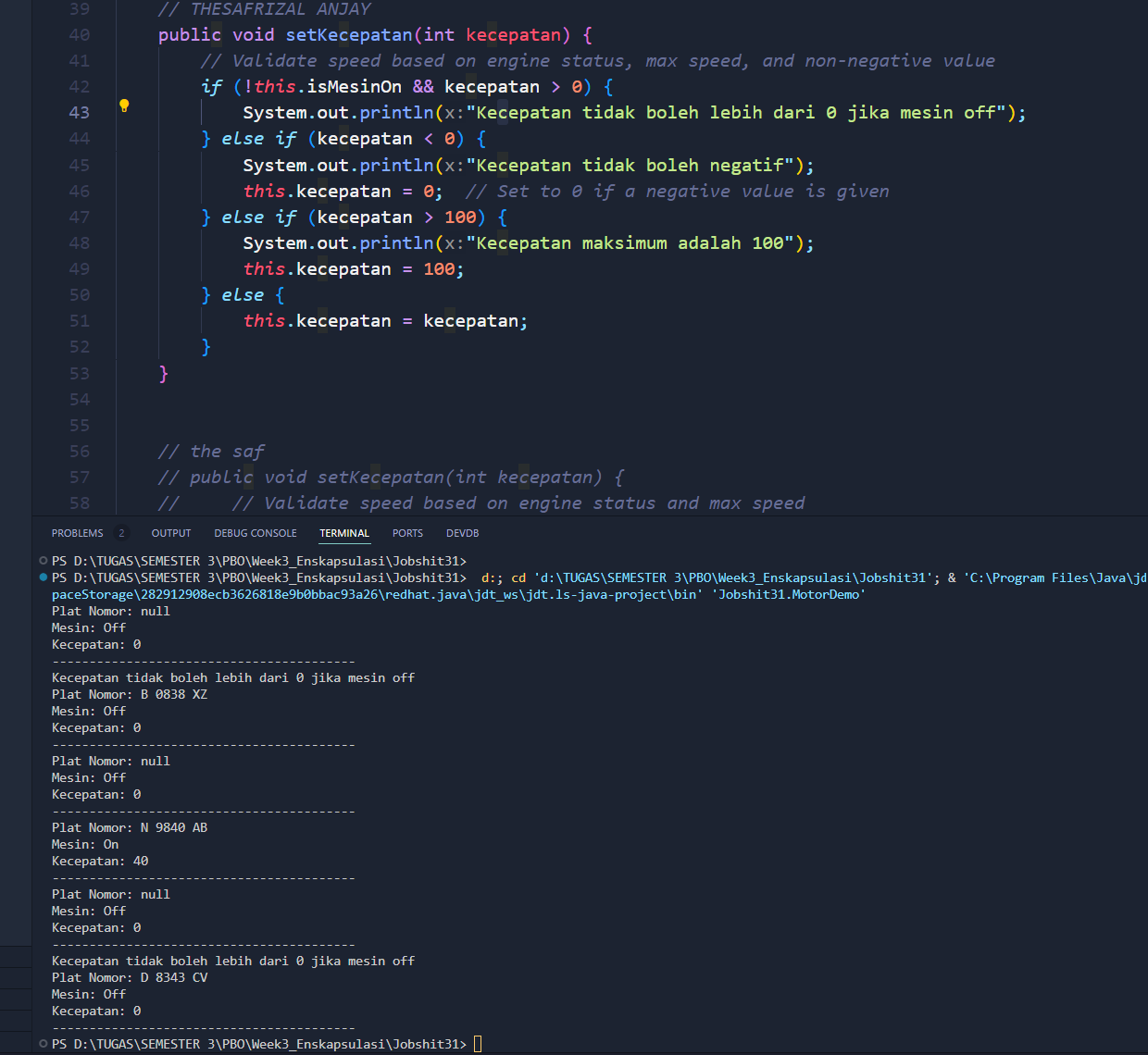
Getter: Retrieves the value of a private attribute. It allows external code to access the value without directly modifying it.

Setter: Allows external code to modify the value of a private attribute while also potentially performing validation or logic before updating the value.

1. Change the class of the Motor so that the maximum speed is 100



1. Change the class of the motorcycle so that the speed should not be negative



## 3.4 Experiment 3 - Constructor

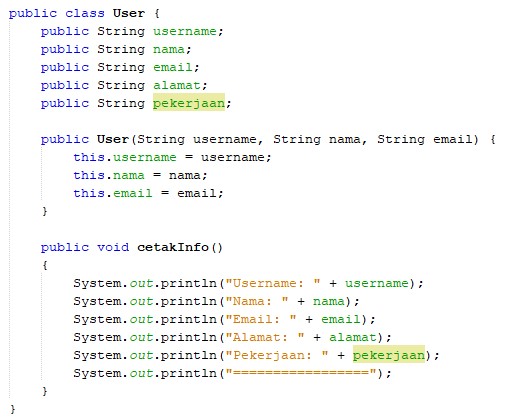
In the previous lesson, object instantiation of a class was done using **the new syntax**

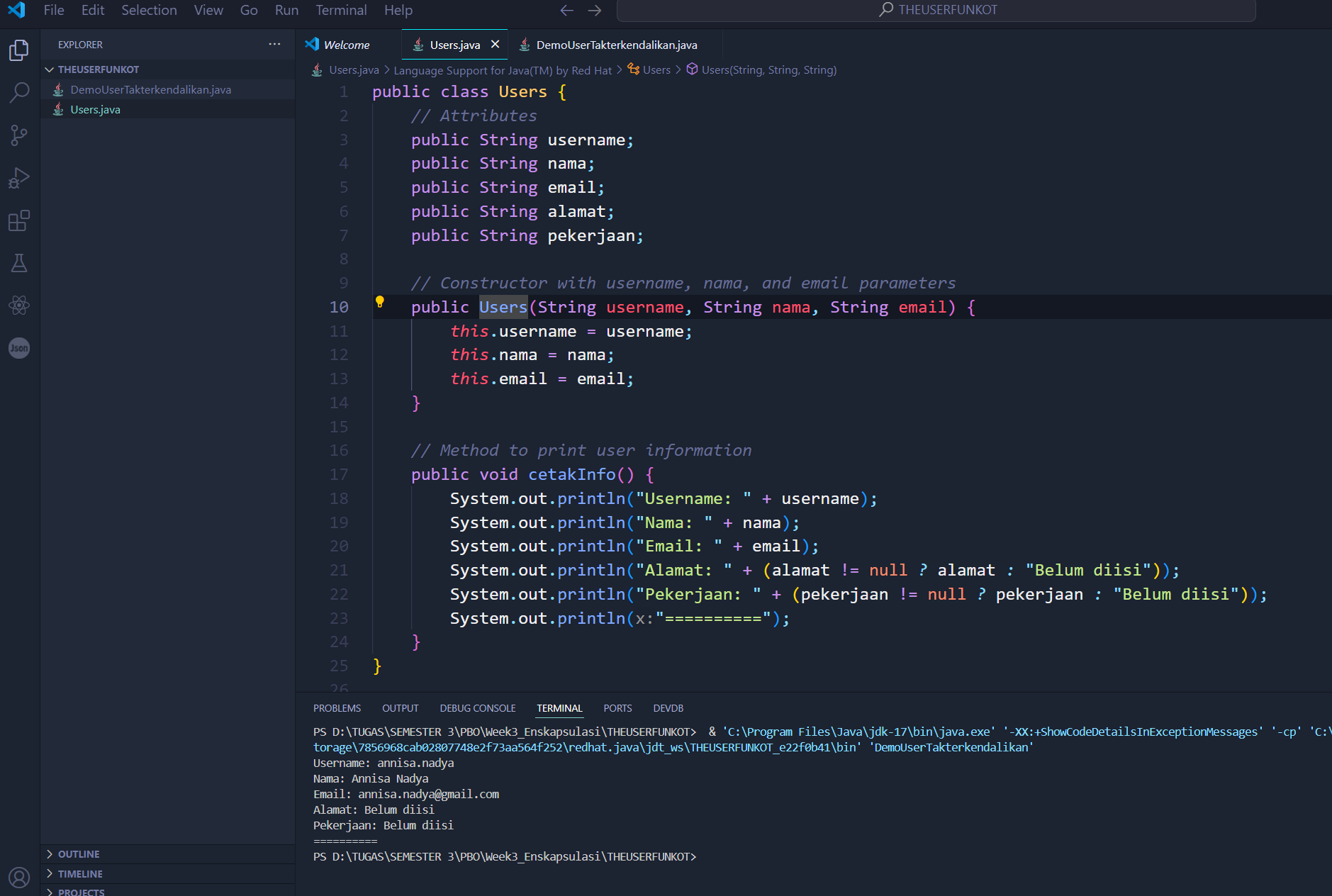
**<NameClass>();** e.g. motor1 = new Motor();

With that line of code, we've used the default constructor Motor() without any parameters. Therefore, any attribute value on motor1 will have a default value. Brand attributes of type string have a default value of **null**, the isMachineOn attribute of type is of type boolean with a default value **of false**, and speed attributes of type integer have a default value of **0**.

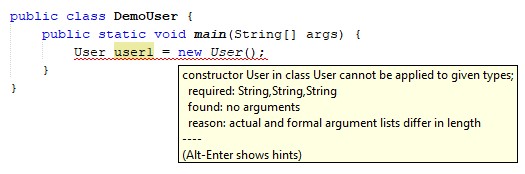
In some cases, we want an object of a given class to already have a value for some (or all) of its attributes by the time the object is created.

1. For example, in an information system, there is a User class that has the attributes of username, name, email, address, and occupation. When a user object is created, it must already have username, name, and email values. With this need, we have to create a new constructor as follows:

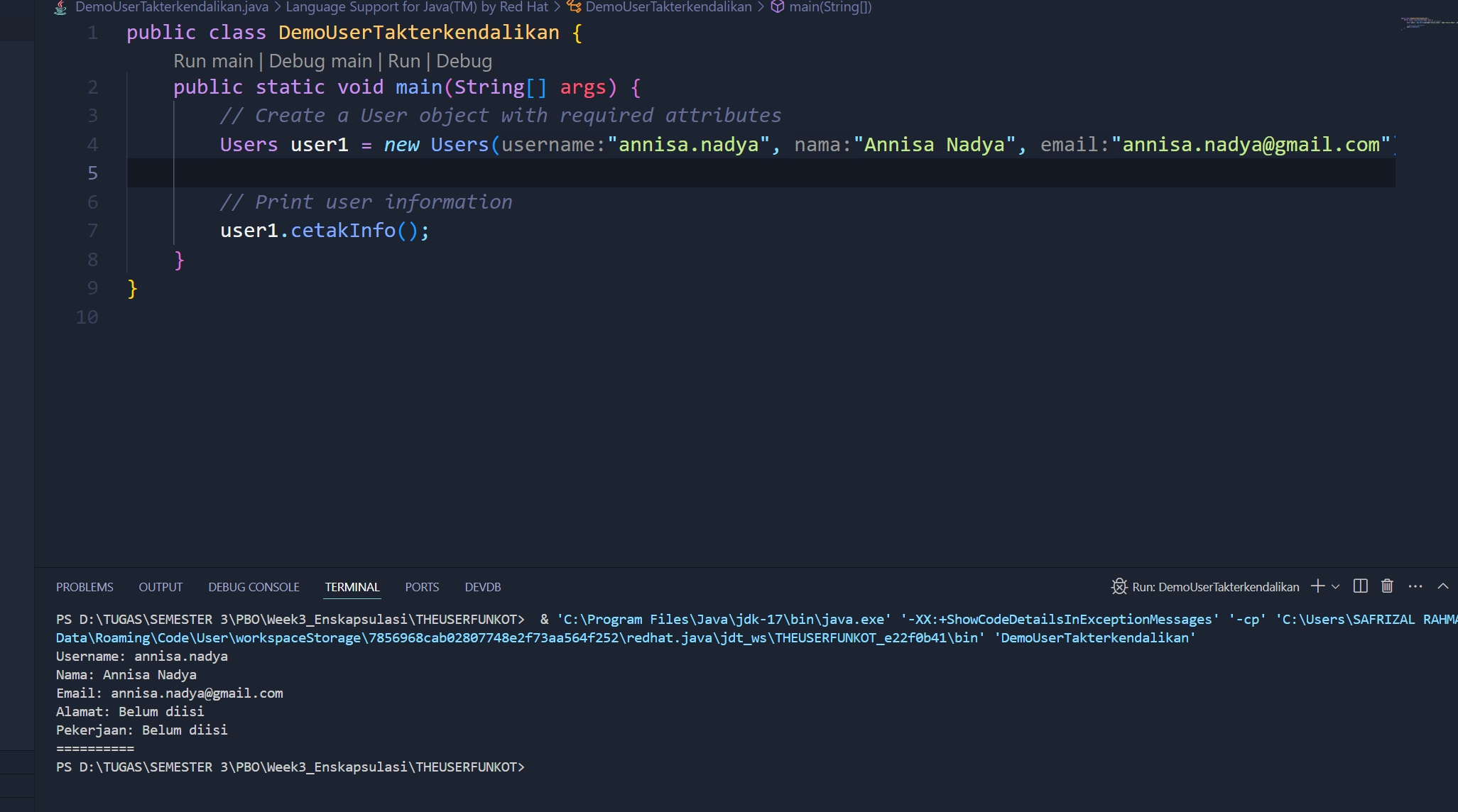




1. Once we provide a new constructor explicitly, the default constructor User() can no longer be used unless we create it as well. Multiple constructors will be discussed in overloading and overriding material.

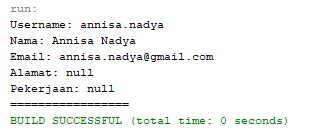
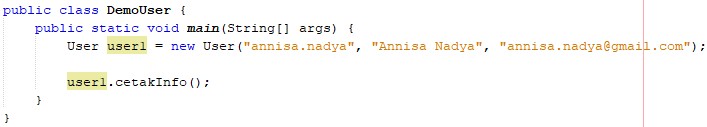


1. Instantiating a new user object with the constructor that has been created in no. 1 can be done in the following way:



4.

The results are as follows:



## 3.5 Questions

1. What is a constructor?

A constructor is a special method in a class that is called when a class object is created.

Its main purpose is to initialize object attributes.

The constructor can accept parameters that allow you to set initial values ​​for attributes when instantiating an object.

1. What are the rules for creating constructors?

- The constructor must have the same name as the class.

- The return type cannot contain even 'cancel'.

- Can accept parameters to initialize attributes or have no parameters (default constructor).

- If a class does not have a defined constructor, Java provides a default constructor.

Defining a custom constructor will disable use of the default constructor unless explicitly stated.

- The constructor can be overloaded.

h.

You can have multiple constructors with different parameters in one class.

1. Do an analysis and make a conclusion whether the constructor can be private?

Yes, constructors can be private.

Private constructors restrict the creation of objects from outside the class.

This is usually used when: - Single pattern: A class can only have one instance.

A private constructor ensures that no other class instances can be created outside of the class itself.

- Factory Methods: Classes can use static factory methods to control object creation.

- Additional classes: Classes that only contain static methods (such as "Math" in Java) can have private constructors that prevent instantiation, as there is no need to create a class object.

Conclusion: Yes, constructors can be private and are often used for design patterns or utility purposes that require instantiating objects to control or constrain.

# 4. Duties

1. In a savings and loan cooperative information system, there is a member class that has attributes such as ID card number, name, borrowing limit, and loan amount. Members can borrow money with a specified borrowing limit. Members can also repay the loan in installments. When the Member installs the loan, the loan amount will be reduced according to the nominal amount paid in installments.

Create the Member class, assign attributes, methods and constructors as needed. Test with the following TestKcooperative to check if the Member class you created is as expected.

*Note that the value of the loan attribute cannot be changed randomly from outside the class, but can only be changed through the loan() and installment() methods.*

public class TestCooperative

{

public static void main(String[] args)

{

Member1 = new Member("111333444", "Donny", 5000000);

System.out.println("Member Name: " + member1.getName());

System.out.println("Loan Limit: " + member1.getLimitLoan());

System.out.println("\nBorrow 10,000,000..."); member1.borrow(10000000);

System.out.println("Current loan amount: " + member1.getLoan Amount());

System.out.println("\nBorrow 4,000,000..."); member1.borrow(4000000);

System.out.println("Current loan amount: " + member1.getLoan Amount());

System.out.println("\nPaying 1,000,000 installments");

Member1.Installment(1000000);

System.out.println("Current loan amount: " + member1.getLoan Amount());

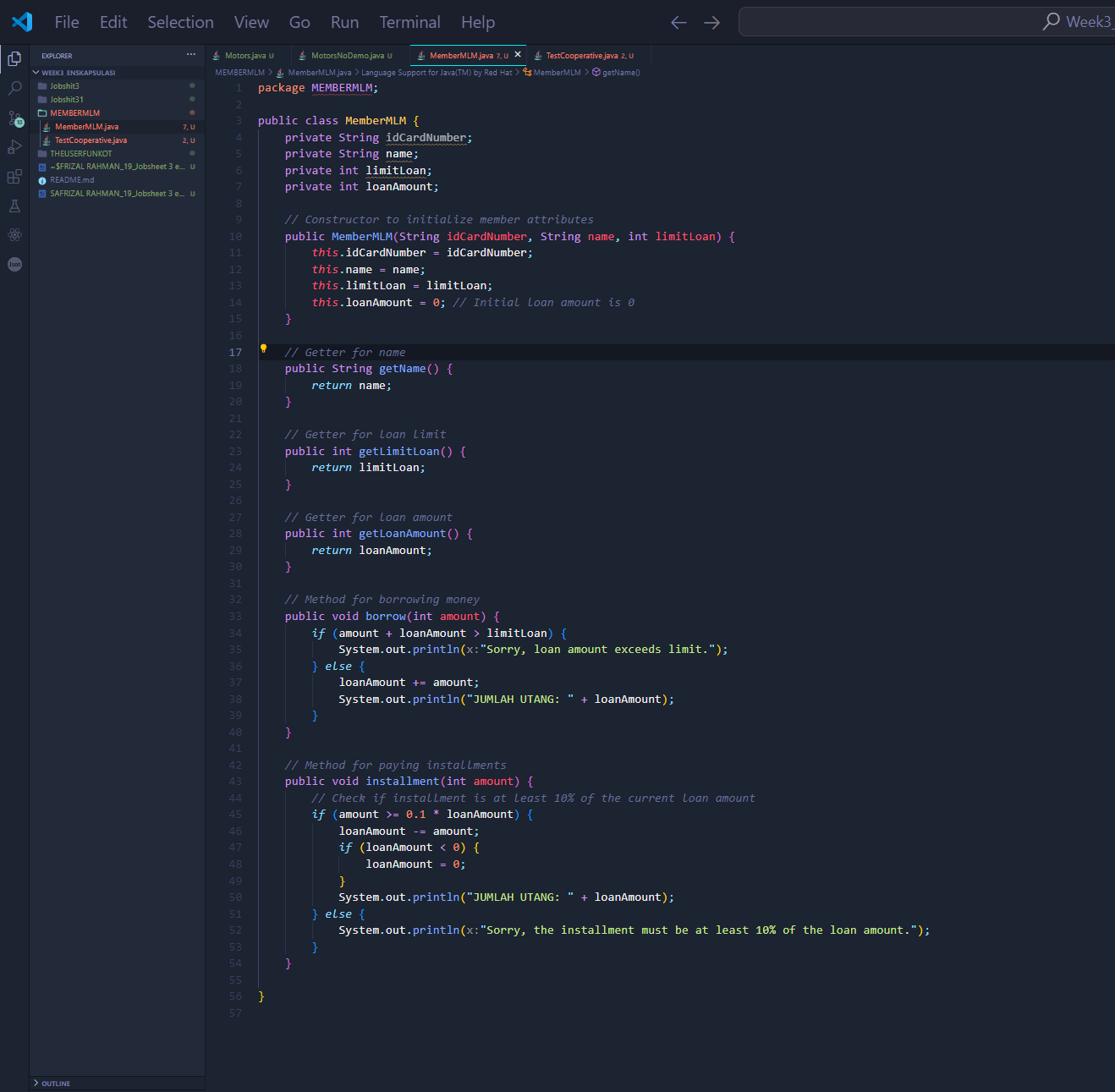
System.out.println("\nPaying 3,000,000 installments");

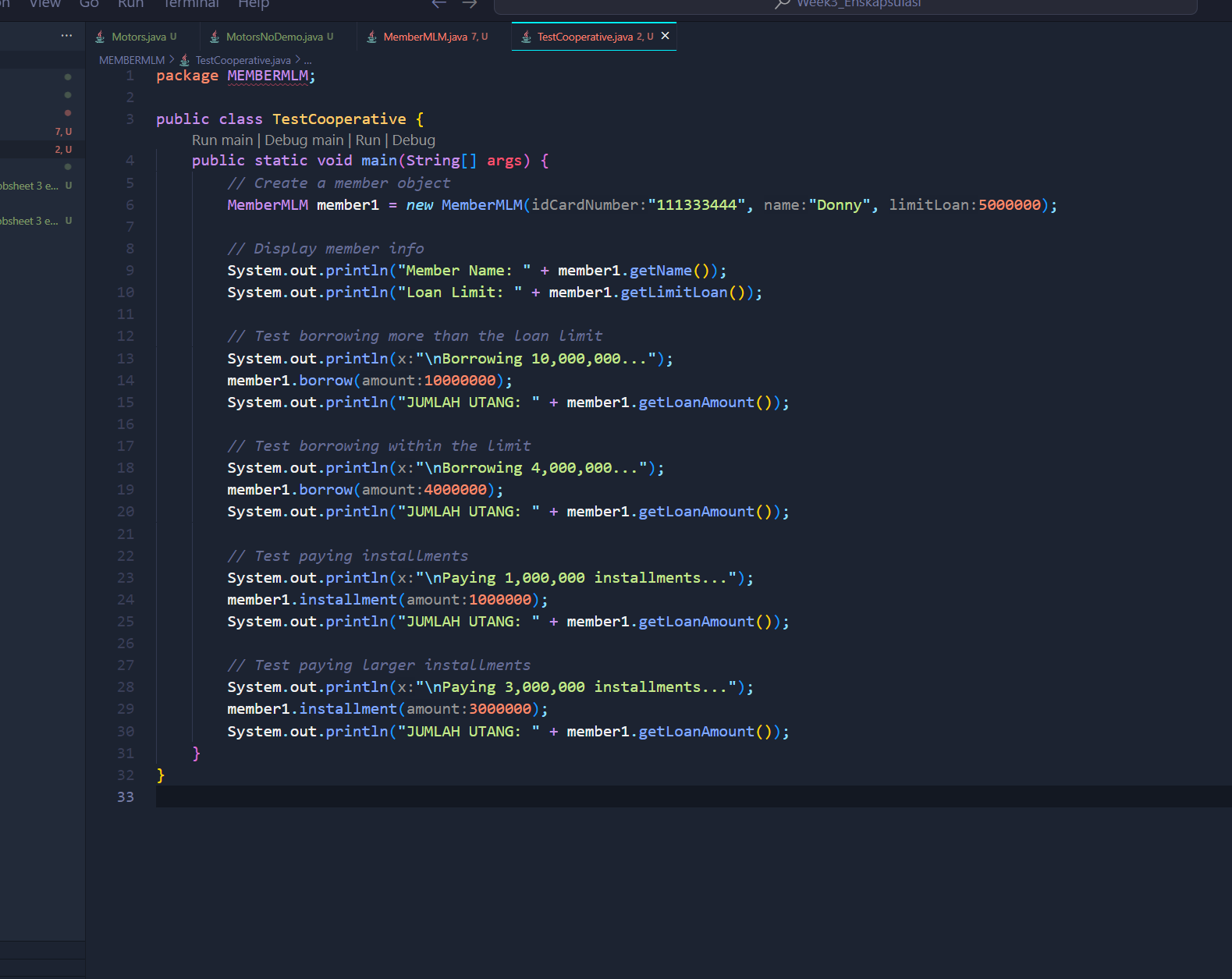
Member1.installment(3000000);

System.out.println("Current loan amount: " + member1.getLoan Amount());

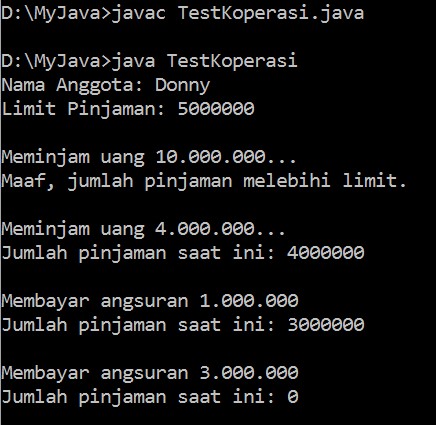
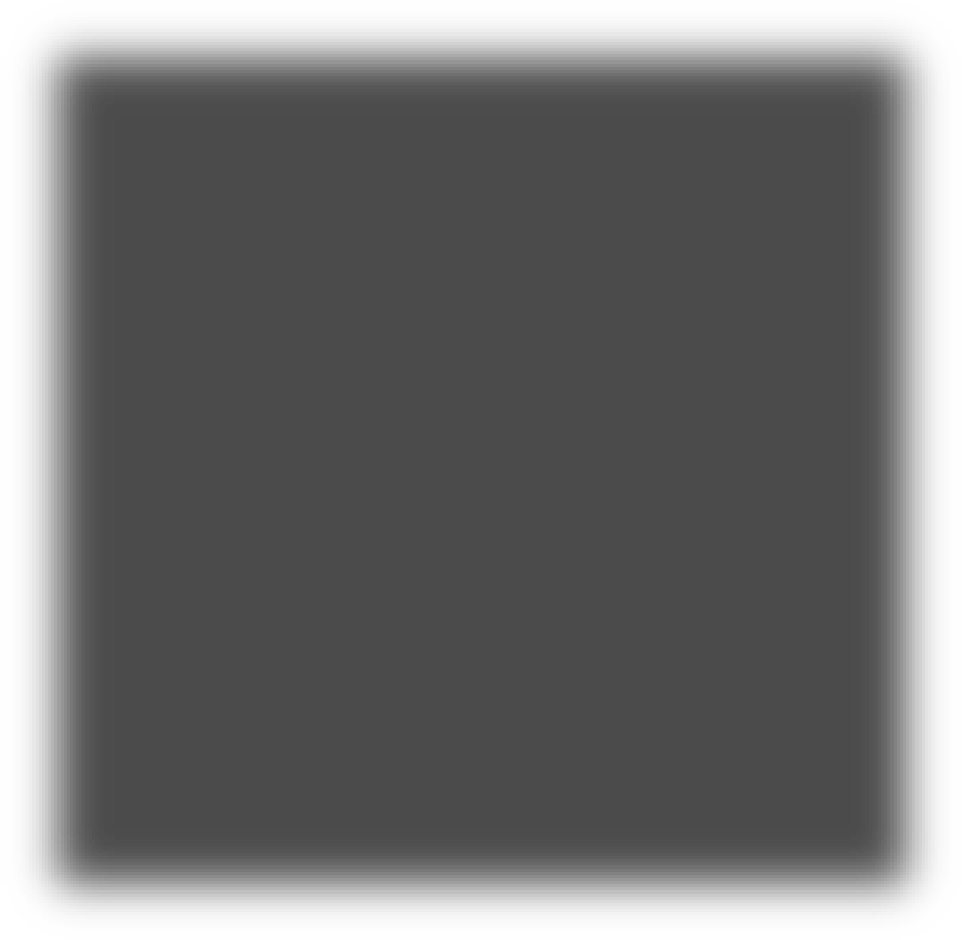
}

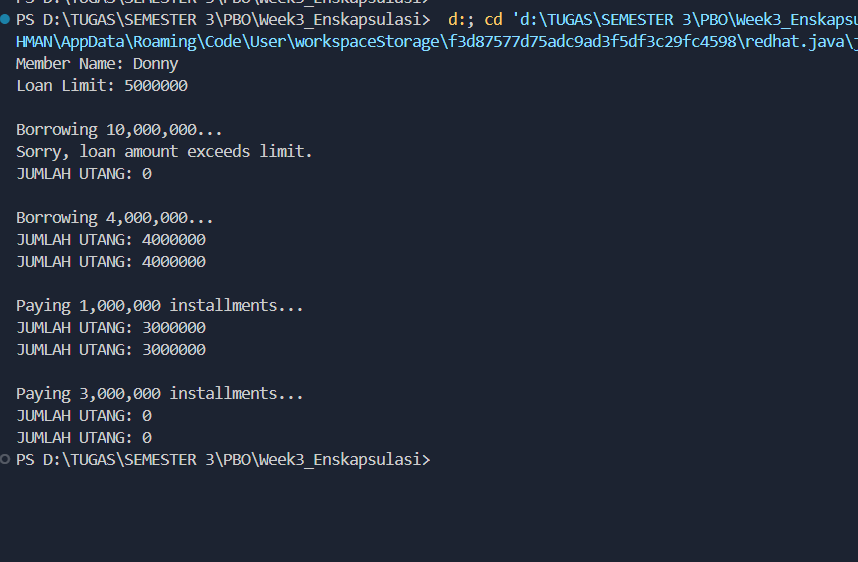
}





Expected results:





2. Modify the Member class so that the nominal amount that can be paid in installments is at least 10% of the current loan amount. If the installment is less than that, then a warning appears "Sorry, the installment must be 10% of the loan amount".

