VERSI 0.1

October 2025

ADVANCED ROGRAMMING

MODUL 2 - SIMPLE REFACTORING

WRITTEN BY :

Ir. Wildan Suharso, M.Kom.

Muhammad Ega Faiz Fadlillah

M. Ramadhan Titan Dwi .C

INFORMATICS LABORATORY TEAM

UNIVERSITY OF MUHAMMADIYAH MALANG

INTRODUCTION

OBJECTIVES

1. Students are able to understand refactoring
2. Students are able to understand refactoring techniques
3. Students are able to perform/implement refactoring

MODULE TARGETS

1. Students can understand the concept of refactoring
2. Students are able to implement the concept of refactoring

PREPARATION

1. Java Development Kit
2. Text Editor / IDE (**Preferably** Intellij IDEA, Netbeans, etc).

KEYWORDS

Refactoring

TABLE OF CONTENTS

THEORY

REFACTORING

A. What is Refactoring

Code refactoring is the process of improving the internal structure of source code without changing how it works. In other words, refactoring does not add new features, but focuses on refining the code to be cleaner, more efficient, and easier to maintain.

The main reason we do refactoring is to make the code better. When code is clean and well-organized, developers can work faster, make fewer mistakes, and it's easier to take care of the code in the future.

B. When to Do Refactoring

- **Before Adding New Features**

  Refactoring before adding new features is very important to ensure the existing code structure is in optimal condition. With clean and organized code, new features can be integrated more easily and efficiently.

- **When Fixing Bugs**

  Refactoring when fixing bugs helps find the root problem faster. With cleaner code, the fixing process becomes easier and reduces the chance of creating new bugs.

- **Dealing with Code Smells**

  Code smells are signs that there may be problems in the code, such as methods that are too long, duplicate code, or unclear variable names. By doing refactoring, you can remove unnecessary confusion and complexity, making the code easier to understand and maintain.

- **Through the Code Review Process**

  Code review is an evaluation process where code written by one developer is checked by another developer or a team. This process aims to ensure that the code meets certain quality standards.

C. Why Refactoring Is Necessary

Refactoring is the process of simplifying and improving the structure of code without changing what it does. The goal is to make the code easier to understand and more concise. Here are some reasons why refactoring is very important:

- **Improve Code Understanding:** Refactoring makes the code clearer, so other developers can more easily understand what the code is doing.
- **Help Find and Fix Bugs:** By simplifying the code, refactoring makes it easier for developers to find and fix possible errors.
- **Speed Up Development:** More organized and simple code allows developers to work faster, making the software development process more efficient.
- **Improve Software Design:** Refactoring helps improve the overall quality of the software design, making it easier to maintain and build on in the future.

D. Refactoring Techniques

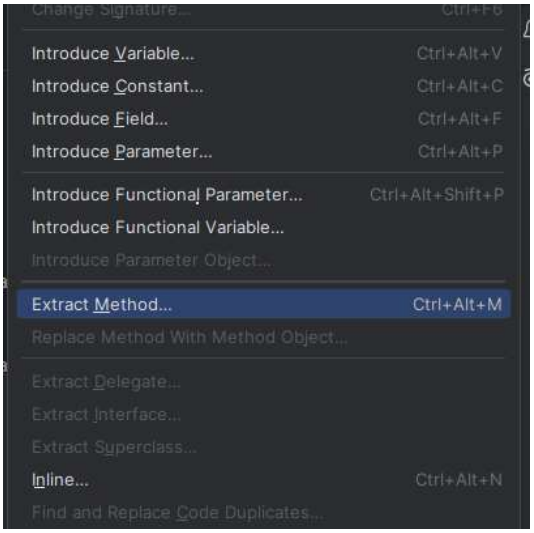
Here are some common refactoring techniques :

1. Extract Method

Extract Method is used to break a long, complex method into smaller parts. This technique moves a specific part of the code into a new method, so the main code becomes shorter, more structured, and easier to understand.

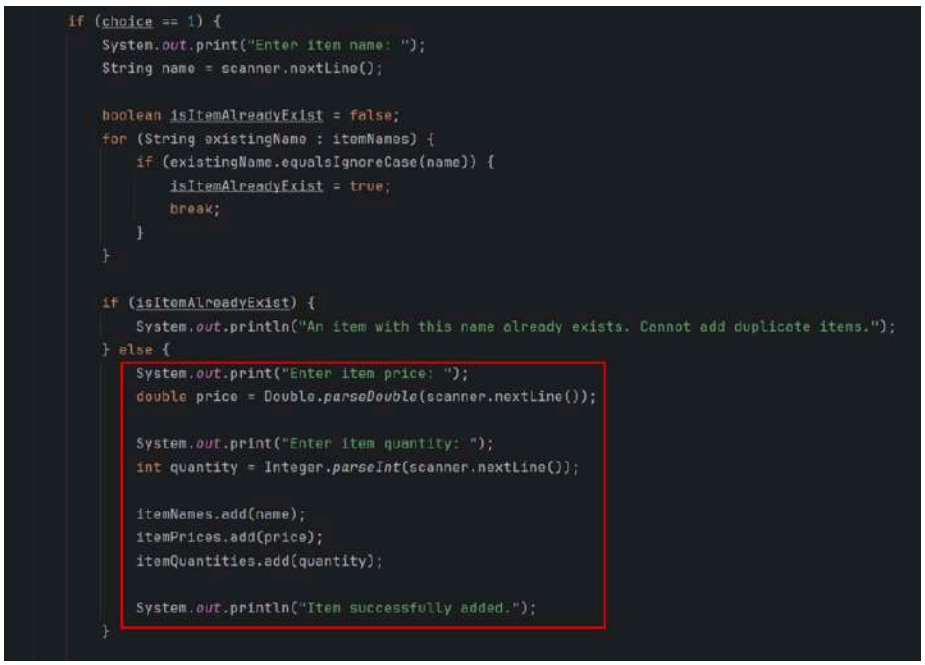
How to Do Extract Method :

- Select the block of code you want to separate
- Right-click the code, choose **Refactor → Extract Method**
- Give the new method a name that matches its function or purpose



a. Before

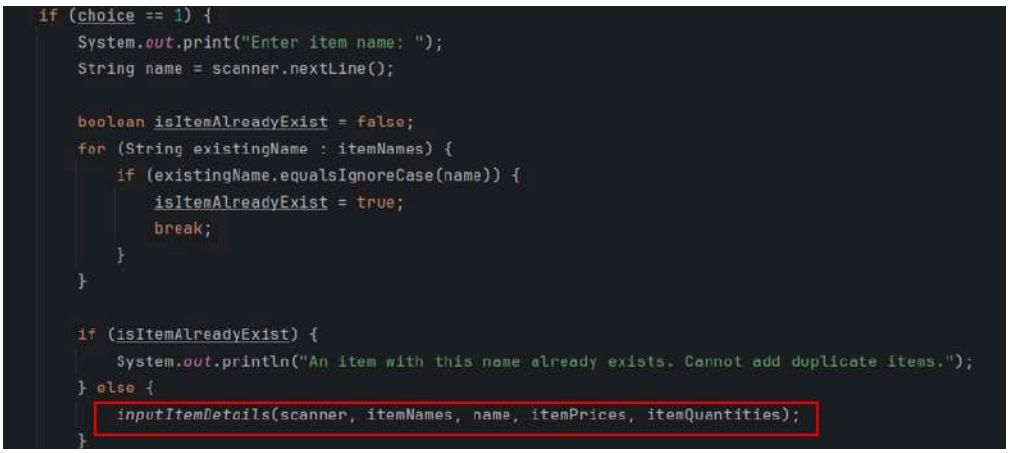
In the example below, there is an if-else structure where option 1 is used to add an item. However, the process of adding the item is written directly inside the case block, making the code long and less structured.



```
if (choice == 1) {
    System.out.print("Enter item name: ");
    String name = scanner.nextLine();

    boolean isItemAlreadyExist = false;
    for (String existingName : itemNames) {
        if (existingName.equalsIgnoreCase(name)) {
            isItemAlreadyExist = true;
            break;
        }
    }

    if (isItemAlreadyExist) {
        System.out.println("An item with this name already exists. Cannot add duplicate items.");
    } else {
        System.out.print("Enter item price: ");
        double price = Double.parseDouble(scanner.nextLine());

        System.out.print("Enter item quantity: ");
        int quantity = Integer.parseInt(scanner.nextLine());

        itemNames.add(name);
        itemPrices.add(price);
        itemQuantities.add(quantity);

        System.out.println("Item successfully added.");
    }
```

b. After

After applying Extract Method, the block of code that handles entering item details like **product name, item price, and item quantity** is refactored into a new method called **inputItemDetails()**.



```
if (choice == 1) {
    System.out.print("Enter item name: ");
    String name = scanner.nextLine();

    boolean isItemAlreadyExist = false;
    for (String existingName : itemNames) {
        if (existingName.equalsIgnoreCase(name)) {
            isItemAlreadyExist = true;
            break;
        }
    }

    if (isItemAlreadyExist) {
        System.out.println("An item with this name already exists. Cannot add duplicate items.");
    } else {
        inputItemDetails(scanner, itemNames, name, itemPrices, itemQuantities);
    }
```

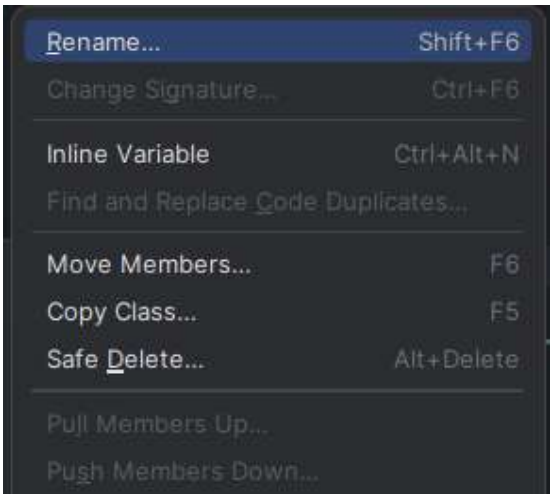
```
    private static void inputItemDetails(Scanner scanner, ArrayList<String> itemNames, String name,   1 usage
                                         ArrayList<Double> itemPrices, ArrayList<Integer> itemQuantities) {
        System.out.print("Enter item price: ");
        double price = Double.parseDouble(scanner.nextLine());

        System.out.print("Enter item quantity: ");
        int quantity = Integer.parseInt(scanner.nextLine());

        itemNames.add(name);
        itemPrices.add(price);
        itemQuantities.add(quantity);

        System.out.println("Item successfully added.");
    }
}
```

2. Rename Method/Variable

The Rename Method/Variable technique is the process of changing the name of an existing method or variable to a more accurate name, and automatically updating all related references across the code. This way, we don't need to manually change each place where it's used.

How to Do Rename Method/Variable :

- Select the variable or method you want to rename.
- Right-click the variable or method → choose **Refactor** → **Rename**
- Enter the new name for the method/variable.



a. Before

In the example below, there is a variable named **result** used to store the salary value after tax is deducted. However, this variable name is not descriptive, which can confuse readers about its purpose.

```java
package RefactorExample;

public class RefactorExampleV2 {
    public static void main(String[] args) {
        double baseSalary = 3000.0;
        double taxRate = 0.15;

        double result = calculateSalaryAfterTax(baseSalary, taxRate);
        double finalSalary = applyBonus(result);

        System.out.println("Base Salary: $" + baseSalary);
        System.out.println("Tax Rate: " + taxRate);
        System.out.println("Salary After Tax: $" + result);
        System.out.println("Final Salary After Bonus: $" + finalSalary);
    }

    public static double calculateSalaryAfterTax(double salary, double taxRate) {
        return salary - (salary * taxRate);
    }

    public static double applyBonus(double salary) { 1 usage
        return salary + 500;
    }
}
```

b. After

After refactoring, the variable name result is changed to **salaryAfterTax**, which is a more descriptive name.

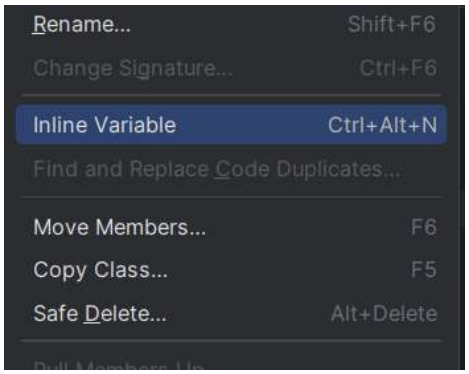
```java
public class Salary {
    public static void main(String[] args) {
        double baseSalary = 3000.0;
        double taxRate = 0.15;

        double salaryAfterTax = calculateSalaryAfterTax(baseSalary, taxRate);
        double finalSalary = applyBonus(salaryAfterTax);

        System.out.println("Base Salary: $" + baseSalary);
        System.out.println("Tax Rate: " + taxRate);
        System.out.println("Salary After Tax: $" + salaryAfterTax);
        System.out.println("Final Salary After Bonus: $" + finalSalary);
    }

    public static double calculateSalaryAfterTax(double salary, double taxRate) {
        return salary - (salary * taxRate);
    }

    public static double applyBonus(double salary) { 1 usage
        return salary + 500;
    }
}
```

3. Inline Variable

Inline Variable is used to replace a variable that has a simple value and is used only once by writing the value or expression directly where it's used. The main goal is to make the code shorter and easier to read, and to reduce complexity from variables that aren't really needed.

How to Do Inline Variable

- Choose a variable that is used only once and has a simple value.
- Right-click the variable → choose **Refactor** → **Inline Variable**.



a. Before

In the example below, the variable **area** is used only once and stores a simple calculation for the area of a circle. Because it's used only once, we can simplify it.

```java
import java.util.Scanner;

public class CircleAreaCalculator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the radius of the circle: ");
        double radius = scanner.nextDouble();

        final double PHI = 3.14;
        double area = PHI * radius * radius;

        System.out.println("\n--- Result ---");
        System.out.println("Radius: " + radius);
        System.out.println("Area: " + area);

        scanner.close();
    }
}
```

b. After

After refactoring, the variable area that stores the expression **3.14 * radius * radius** is removed. The expression is written directly where the area variable was used before, or inside **System.out.println()**.

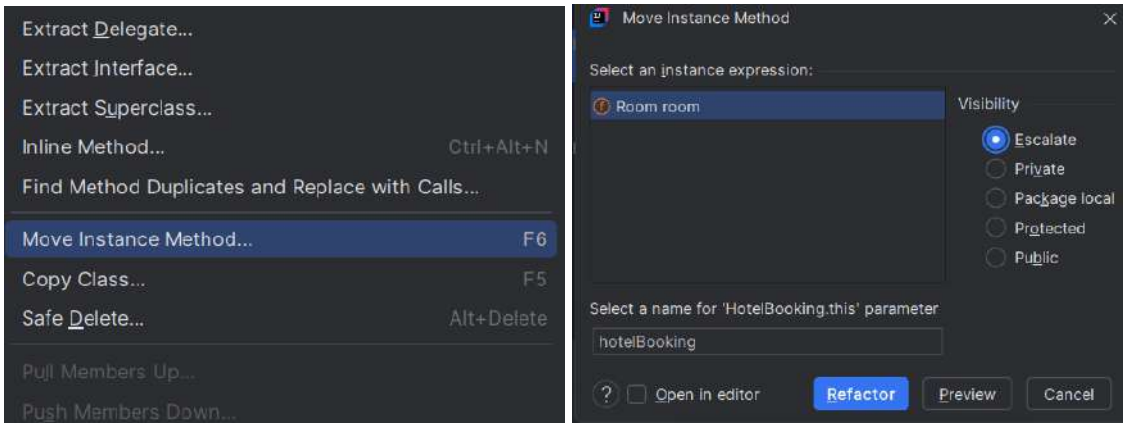
```
import java.util.Scanner;

public class CircleAreaCalculator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the radius of the circle: ");
        double radius = scanner.nextDouble();

        System.out.println("\n--- Result ---");
        System.out.println("Radius: " + radius);
        System.out.println("Area: " + 3.14 * radius * radius);

        scanner.close();
    }
}
```

4. Move Method/Field

Move Method or Move Field is a refactoring technique where a method or a field (attribute) is moved from one class to another class that is more relevant to its function. The goal is to make the code more structured, modular, and easier to maintain.

How to Do Move Method/Field

- Choose a method you want to move to another class.
- Right-click → choose **Refactor** → **Move Instance Method**.
- Select the destination class where the method or field will be moved.
- If moving a method, give a new name to the object or method if needed.



a. Before

At first, the HotelBooking class has the methods **calculateTotalPrice()** and **calculateFinalPrice()**, but their placement is not ideal because the logic is more related to room properties, such as price per night.

```
class Room {
    private String roomType;
    private double pricePerNight;

    public Room(String roomType, double pricePerNight){
        this.roomType = roomType;
        this.pricePerNight = pricePerNight;
    }

    public String getRoomType(){
        return roomType;
    }

    public double getPricePerNight(){
        return pricePerNight;
    }
}

public class HotelBooking {
    public Room room;
    public int numberOfNights;
    public double taxRate;

    public HotelBooking(Room room, int numberOfNights, double taxRate){
        this.room = room;
        this.numberOfNights = numberOfNights;
        this.taxRate = taxRate;
    }

    public double calculateTotalPrice(){
        return numberOfNights * room.getPricePerNight();
    }

    public double calculateFinalPrice(){
        return calculateTotalPrice() + (calculateTotalPrice() * taxRate);
    }

    public void displayDetails(){
        System.out.println("Room Type: " + room.getRoomType());
        System.out.println("Number of Nights: " + numberOfNights);
        System.out.println("Price per Night: $ " + room.getPricePerNight());
        System.out.println("Total Price: $ " + calculateTotalPrice());
        System.out.println("Final Price: $ " + calculateFinalPrice());
    }
}
```

b. After

After refactoring, we move both methods to the Room class, so they are now in a more relevant place that deals with rooms.

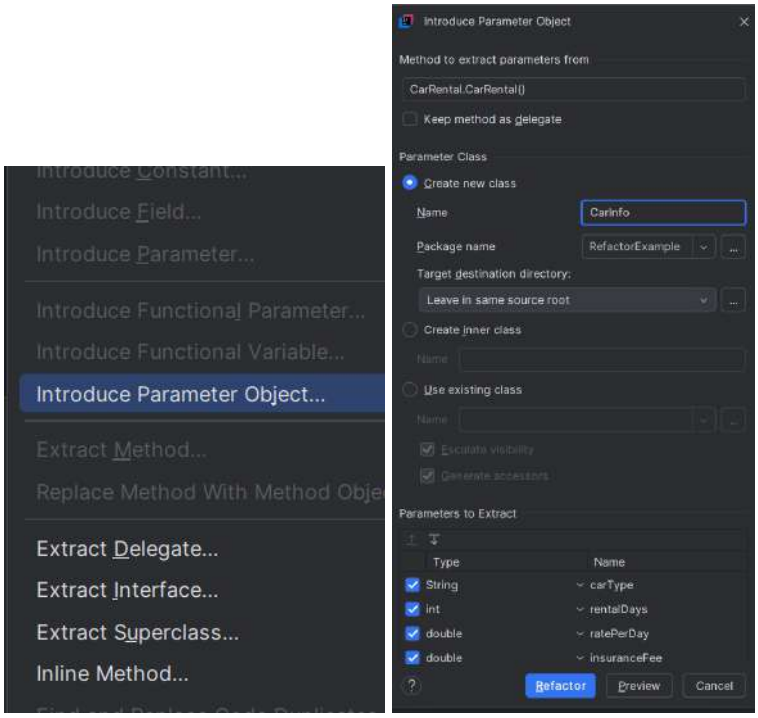
```
class Room {
    private String roomType;
    private double pricePerNight;

    public Room(String roomType, double pricePerNight){
        this.roomType = roomType;
        this.pricePerNight = pricePerNight;
    }

    public String getRoomType(){
        return roomType;
    }

    public double getPricePerNight(){
        return pricePerNight;
    }

    public double calculateFinalPrice(HotelBooking hotelBooking){
        return
          hotelBooking.room.calculateTotalPrice(hotelBooking) +
          (hotelBooking.room.calculateTotalPrice(hotelBooking)
            * hotelBooking.taxRate);
    }

    public double calculateTotalPrice(HotelBooking hotelBooking){
        return hotelBooking.numberOfNights * getPricePerNight();
    }
}
public class HotelBooking {
    public Room room;
    public int numberOfNights;
    public double taxRate;

    public HotelBooking(Room room, int numberOfNights, double taxRate){
        this.room = room;
        this.numberOfNights = numberOfNights;
        this.taxRate = taxRate;
    }

    public void displayDetails(){
        System.out.println("Room Type: " + room.getRoomType());
        System.out.println("Number of Nights: " + numberOfNights);
        System.out.println("Price per Night: $ " + room.getPricePerNight());
        System.out.println("Total Price: $ " + room.calculateTotalPrice(this));
        System.out.println("Final Price: $ " + room.calculateFinalPrice(this));
    }
}
```

5. Introduce Parameter Object

This technique is used to simplify a method that has a group of parameters that often appear together. Instead of repeatedly writing many similar parameters in the method, we combine those parameters into a dedicated class (an object). This way, the method that previously had many parameters now only takes one object as a parameter.

How to Do Introduce Parameter Object

- Choose a method whose parameters you want to turn into an object.
- Right-click → choose **Refactor** → **Introduce Parameter Object**.
- Name the new class and select which parameters to include.



a. Before

In the methods **CarRental()** and **calculateTotalCost(),** there are similar parameters: **int rentalDays, double ratePerDay, and double insuranceFee**. Here we will use the introduce parameter object refactoring.

```
public class CarRental {
    private final String carType;
    private final int rentalDays;
    private final double ratePerDay;
    private final double insuranceFee;

    public CarRental(String carType, int rentalDays, double ratePerDay, double insuranceFee) {
        this.carType = carType;
        this.rentalDays = rentalDays;
        this.ratePerDay = ratePerDay;
        this.insuranceFee = insuranceFee;
    }

    public double calculateTotalCost(int rentalDays, double ratePerDay, double insuranceFee) {
        double rentalCost = rentalDays * ratePerDay;
        return rentalCost + insuranceFee;
    }

    public void displayDetails() {
        double totalCost = calculateTotalCost(rentalDays, ratePerDay, insuranceFee);
        System.out.println("Car Type: " + carType);
        System.out.println("Rental Days: " + rentalDays);
        System.out.println("Rate Per Day: $" + ratePerDay);
        System.out.println("Insurance Fee: $" + insuranceFee);
        System.out.println("Total Cost: $" + totalCost);
    }

    public static void main(String[] args) {
        CarRental rental = new CarRental("SUV", 4, 75.0, 30.0);
        rental.displayDetails();
    }
}
```

b. After

After introducing a parameter object, we create a new record class called **CarInfo** that contains the parameters **int rentalDays, double ratePerDay,** and **double insuranceFee**. A record class is designed to make it easy to create classes that only store data (data carrier/data holder).

```
package RefactorExample;

public record CarInfo(int rentalDays, double ratePerDay, double insuranceFee) {
}
```

In the **CarRental()** constructor, which originally had the parameters **int rentalDays, double ratePerDay,** and **double insuranceFee**, it will change to a single parameter **CarInfo carInfo** because of the introduce parameter object refactor.

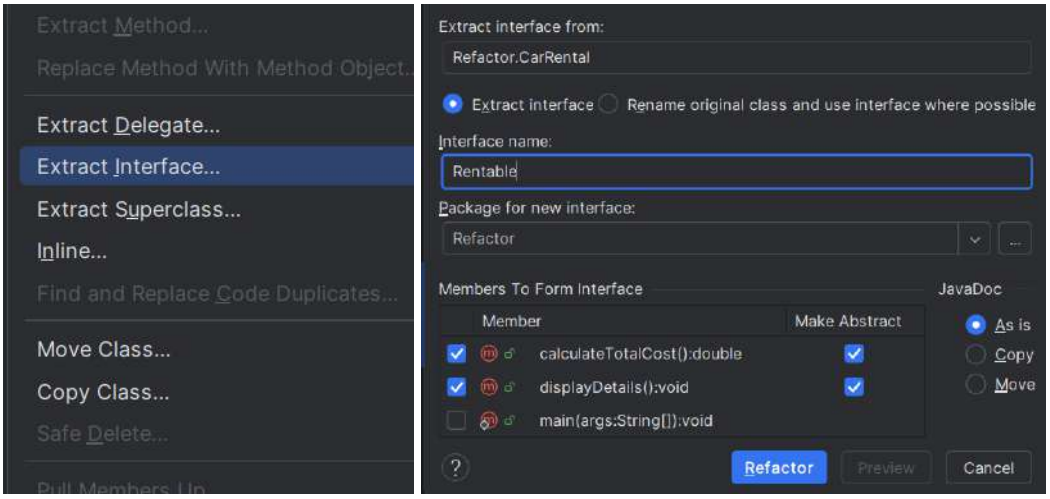
```
public class CarRental {
    private final String carType;
    private final int rentalDays;
    private final double ratePerDay;
    private final double insuranceFee;

    public CarRental(String carType, CarInfo carInfo) {
        this.carType = carType;
        this.rentalDays = carInfo.rentalDays();
        this.ratePerDay = carInfo.ratePerDay();
        this.insuranceFee = carInfo.insuranceFee();
    }

    public double calculateTotalCost(CarInfo carInfo) {
        double rentalCost = carInfo.rentalDays() * carInfo.ratePerDay();
        return rentalCost + carInfo.insuranceFee();
    }

    public void displayDetails() {
        double totalCost = calculateTotalCost(new CarInfo(rentalDays, ratePerDay, insuranceFee));
```

6. Extract Interface

Extract Interface is a technique used to create an interface from an existing class. It helps with abstraction and makes it easier to create different class variations.

How to Do Extract Interface :

- Choose the methods you want to include in an interface.
- Right-click → choose **Refactor** → **Extract Interface**.
- Name the interface and select which methods to include.



a. Before

In the **CarRental class**, there are methods like **calculateTotalCost()** and **displayDetails().** However, this class has no abstraction, so it's hard to create other rental types (like LuxuryCarRental, ElectricCarRental, or MotorcycleRental) without changing the class structure.

```
public class CarRental {
    private final String carType;
    private final int rentalDays;
    private final double ratePerDay;
    private final double insuranceFee;

    public CarRental(String carType, int rentalDays, double ratePerDay, double insuranceFee) {
        this.carType = carType;
        this.rentalDays = rentalDays;
        this.ratePerDay = ratePerDay;
        this.insuranceFee = insuranceFee;
    }

    public double calculateTotalCost() {
        double rentalCost = rentalDays * ratePerDay;
        return rentalCost + insuranceFee;
    }

    public void displayDetails() {
        double totalCost = calculateTotalCost();
        System.out.println("Car Type: " + carType);
        System.out.println("Rental Duration: " + rentalDays + " days");
        System.out.println("Rate per Day: $ " + ratePerDay);
        System.out.println("Insurance Fee: $ " + insuranceFee);
        System.out.println("Total Rental Cost: $ " + totalCost);
    }

    public static void main(String[] args) {
        CarRental rental = new CarRental( carType: "SUV", rentalDays: 4, ratePerDay: 75.0, insuranceFee: 50.0);
        rental.displayDetails();
    }
}
```

b. After

After extracting an interface, we create an interface named **Rentable**. This interface contains the methods **calculateTotalCost()** and **displayDetails()**.

```
public interface Rentable {  2 usages  1 implementation
    double calculateTotalCost();  1 usage  1 implementation

    void displayDetails();  1 usage  1 implementation
}
```

The CarRental class will **implement the Rentable interface**, and the two methods calculateTotalCost() and displayDetails() **will be overridden** as part of that implementation.

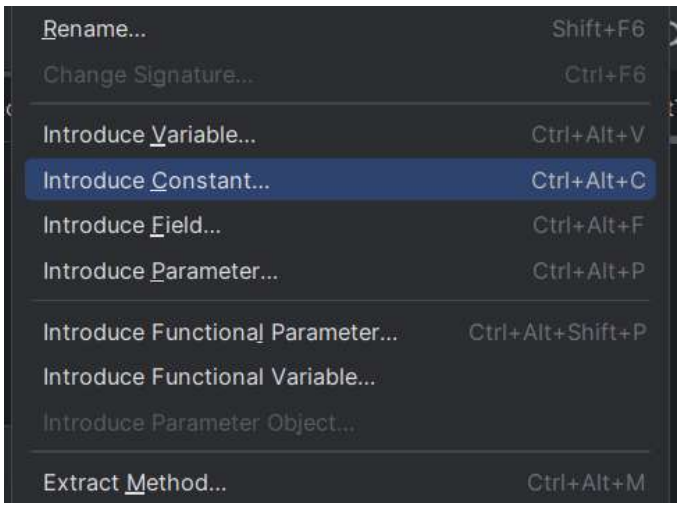
```java
public class CarRental implements Rentable {
    private final String carType;  2 usages
    private final int rentalDays;  3 usages
    private final double ratePerDay;  3 usages
    private final double insuranceFee;  3 usages

    public CarRental(String carType, int rentalDays, double ratePerDay, double insuranceFee) {
        this.carType = carType;
        this.rentalDays = rentalDays;
        this.ratePerDay = ratePerDay;
        this.insuranceFee = insuranceFee;
    }

    @Override  1 usage
    public double calculateTotalCost() {
        return rentalDays * ratePerDay + insuranceFee;
    }

    @Override  1 usage
    public void displayDetails() {
        System.out.println("Car Type: " + carType);
        System.out.println("Rental Duration: " + rentalDays + " days");
        System.out.println("Rate per Day: $" + ratePerDay);
        System.out.println("Insurance Fee: $" + insuranceFee);
        System.out.println("Total Rental Cost: $" + calculateTotalCost());
    }
```

7. Replace Magic Number with Symbolic Constant

This technique **replaces hard-coded numbers** (magic numbers) in your code with named constants. The goal is to make the code easier to understand because the constant's name explains the meaning of the value.

How to Do Replace Magic Number with Symbolic Constant

- Choose a value you want to turn into a constant.
- Right-click → choose **Refactor** → **Introduce Constant**.
- Give the constant a meaningful name.



a. Before

In the CalculateTotalCost() method, there are values **100000** and **0.1**. These are magic numbers because they are written directly with no explanation. This makes it hard to know that 100000 is the minimum spend to get a discount and 0.1 is the discount rate. To make it clearer, replace these with meaningful constants like **DISCOUNT_THRESHOLD** and **DISCOUNT_RATE**.

```
public class ShopTransaction {
    private String itemName;  2 usages
    private int quantity;  3 usages
    private double pricePerItem;  3 usages

    public ShopTransaction(String itemName, int quantity, double pricePerItem) {
        this.itemName = itemName;
        this.quantity = quantity;
        this.pricePerItem = pricePerItem;
    }

    // Method to calculate total cost with discount rule
    public double calculateTotalCost() {  1 usage
        double total = quantity * pricePerItem;

        // Apply discount if total exceeds 100000
        if (total > 100000) {
            double discount = total * 0.1; // 10% discount
            total -= discount;
        }

        return total;
    }
```

b. After

After refactoring, you will have constant variables: **int DISCOUNT_THRESHOLD** with the value 100000 and **double DISCOUNT_RATE with the value 0.1**.

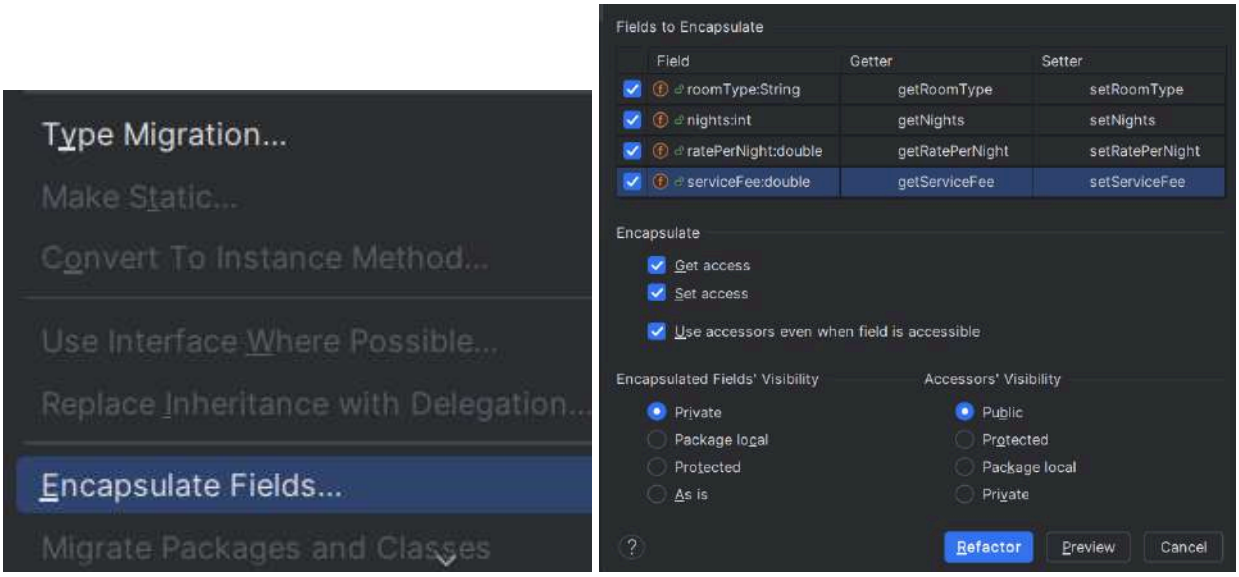
```java
public class ShopTransaction {

    public static final int DISCOUNT_THRESHOLD = 100000;  1 usage
    public static final double DISCOUNT_RATE = 0.1;  1 usage

    private String itemName;  2 usages
    private int quantity;  3 usages
    private double pricePerItem;  3 usages

    public ShopTransaction(String itemName, int quantity, double pricePerItem) {
        this.itemName = itemName;
        this.quantity = quantity;
        this.pricePerItem = pricePerItem;
    }

    // Method to calculate total cost with discount rule
    public double calculateTotalCost() {  1 usage
        double total = quantity * pricePerItem;

        // Apply discount if total exceeds 100000
        if (total > DISCOUNT_THRESHOLD) {
            double discount = total * DISCOUNT_RATE; // 10% discount
            total -= discount;
        }

        return total;
    }
```

8. Encapsulate Field

Encapsulation is an important OOP principle used to protect class attributes so they can't be accessed or changed directly from outside the class. Usually, we make attributes private and provide getter and setter methods to read or update their values.

How to Do Encapsulate Field :

- Choose the attributes you want to encapsulate.
- Right-click → choose **Refactor** → **Encapsulate Fields**
- Select which attributes to encapsulate.



a. Before

The HotelBooking class has four attributes : roomType, nights, ratePerNight, and serviceFee, all with public visibility. This means they can be accessed and modified directly from outside the class, which can cause security issues or data inconsistency.

```
public class HotelBooking {
    String roomType;
    int nights;
    double ratePerNight;
    double serviceFee;

    public HotelBooking(String roomType, int nights) {
        this.roomType = roomType;
        this.nights = nights;

        this.ratePerNight = 100.0;
        this.serviceFee = 50.0;
    }

    public double calculateTotalCost() {
        double roomCost = nights * ratePerNight;
        return roomCost + serviceFee;
    }

    public void displayDetails() {
        double totalCost = calculateTotalCost();
        System.out.println("Room Type: " + roomType);
        System.out.println("Duration: " + nights + " nights");
        System.out.println("Rate per Night: $ " + ratePerNight);
        System.out.println("Service Fee: $ " + serviceFee);
        System.out.println("Total Booking Cost: $ " + totalCost);
    }

    public static void main(String[] args) {
        HotelBooking booking = new HotelBooking( roomType: "Deluxe",  nights: 3);
        booking.displayDetails();
    }
}
```

b. After

After refactoring, these four attributes are changed to private. In return, we provide getter methods to read the values and setter methods to modify them.

```
public class HotelBooking {
    private String roomType;  2 usages
    private int nights;  2 usages
    private double ratePerNight;  2 usages
    private double serviceFee;  2 usages

    public String getRoomType() {  1 usage
        return roomType;
    }

    public void setRoomType(String roomType) {  1 usage
        this.roomType = roomType;
    }

    public int getNights() {  2 usages
        return nights;
    }

    public void setNights(int nights) {  1 usage
        this.nights = nights;
    }

    public double getRatePerNight() {  2 usages
        return ratePerNight;
    }

    public void setRatePerNight(double ratePerNight) {  1 usage
        this.ratePerNight = ratePerNight;
    }

    public double getServiceFee() {  2 usages
```

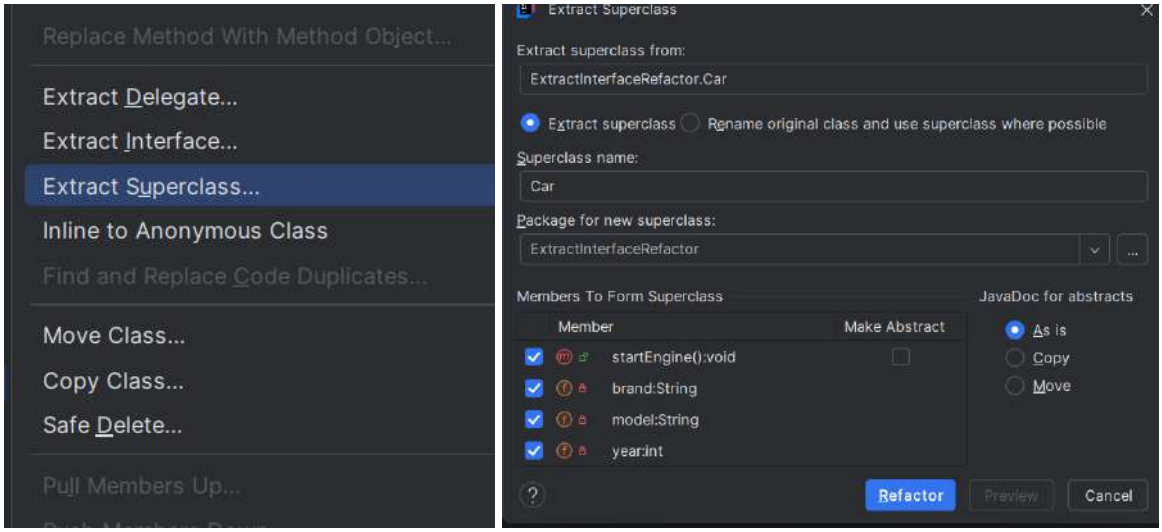
9. Extract Superclass

Extract Superclass is a technique where you create a new parent class (superclass) to hold attributes and methods that are shared by two or more classes. Then, those classes become subclasses and inherit from the superclass. The goal is to reduce duplicate code and make the program structure cleaner.

For example, if you have two classes, Car and Motorcycle, both with brand, model, and year, instead of duplicating these in each class, you can extract a superclass named Vehicle. Then Car and Motorcycle focus on their unique logic (like openTrunk() for cars or popWheelie() for motorcycles), while common data stays in the superclass.

How to Do Extract Superclass

- Choose classes that share common attributes or methods.
- Right-click → choose **Refactor** → **Extract Superclass**.
- Name the new superclass.
- Select which attributes and methods to move to the superclass to avoid duplication. You can also choose to make some methods abstract in the superclass.



a. Before

Initially, there are two classes : Car and Motorcycle. They share common attributes (brand, model, year) and a method startEngine(). This duplication makes the code less efficient and harder to maintain if changes are needed.

```
public class Car {  2 usages
    private String brand;  2 usages
    private String model;  2 usages
    private int year;  2 usages
    private double trunkCapacity;  2 usages

    public Car(String brand, String model, int year, double trunkCapacity) {  1 usage
        this.brand = brand;
        this.model = model;
        this.year = year;
        this.trunkCapacity = trunkCapacity;
    }

    public void startEngine() {  1 usage
        System.out.println("Engine started: " + year + " " + brand + " " + model);
    }

    public void openTrunk() {  1 usage
        System.out.println("Opening trunk with capacity: " + trunkCapacity + " liters");
    }
}
```

```
public class Motorcycle {  2 usages
    private String brand;  4 usages
    private String model;  4 usages
    private int year;  2 usages
    private boolean hasSidecar;  2 usages

    public Motorcycle(String brand, String model, int year, boolean hasSidecar) {  1 usage
        this.brand = brand;
        this.model = model;
        this.year = year;
        this.hasSidecar = hasSidecar;
    }

    public void startEngine() {  1 usage
        System.out.println("Motorcycle engine started: " + year + " " + brand + " " + model);
    }

    public void popWheelie() {  1 usage
        if (!hasSidecar) {
            System.out.println(brand + " " + model + " is popping a wheelie!");
        } else {
            System.out.println(brand + " " + model + " cannot pop a wheelie because it has a sidecar.");
        }
    }
}
```

b. After

After refactoring with extract superclass, we create a new class named Vehicle as the superclass to hold the shared attributes and methods from Car and Motorcycle. Common attributes like brand, model, and year are moved to Vehicle, and the startEngine() method no longer needs to be duplicated in each subclass. Now, the Car class focuses on its unique parts, such as trunkCapacity and the openTrunk() method, while the Motorcycle class keeps its extra attribute hasSidecar and its special method popWheelie().

```
public class Vehicle {  1 usage  1 inheritor
    protected String brand;  2 usages
    protected String model;  2 usages
    protected int year;  2 usages

    public Vehicle(String brand, String model, int year) {  1 usage
        this.brand = brand;
        this.model = model;
        this.year = year;
    }

    public void startEngine() {  1 usage
        System.out.println("Engine started: " + year + " " + brand + " " + model);
    }
}
```

```java
public class Car extends Vehicle {  2 usages
    private double trunkCapacity;  2 usages

    public Car(String brand, String model, int year, double trunkCapacity) {  1 usage
        super(brand, model, year);
        this.trunkCapacity = trunkCapacity;
    }

    public void openTrunk() {  1 usage
        System.out.println("Opening trunk with capacity: " + trunkCapacity + " liters");
    }
}
```

```java
public class Motorcycle extends Vehicle{  2 usages
    private boolean hasSidecar;  2 usages

    public Motorcycle(String brand, String model, int year, boolean hasSidecar) {  1 usage
        super(brand, model, year);
        this.hasSidecar = hasSidecar;
    }

    public void popWheelie() {  1 usage
        if (!hasSidecar) {
            System.out.println(brand + " " + model + " is popping a wheelie!");
        } else {
            System.out.println(brand + " " + model + " cannot pop a wheelie because it has a sidecar.");
        }
    }
}
```

E. How to Do Refactoring

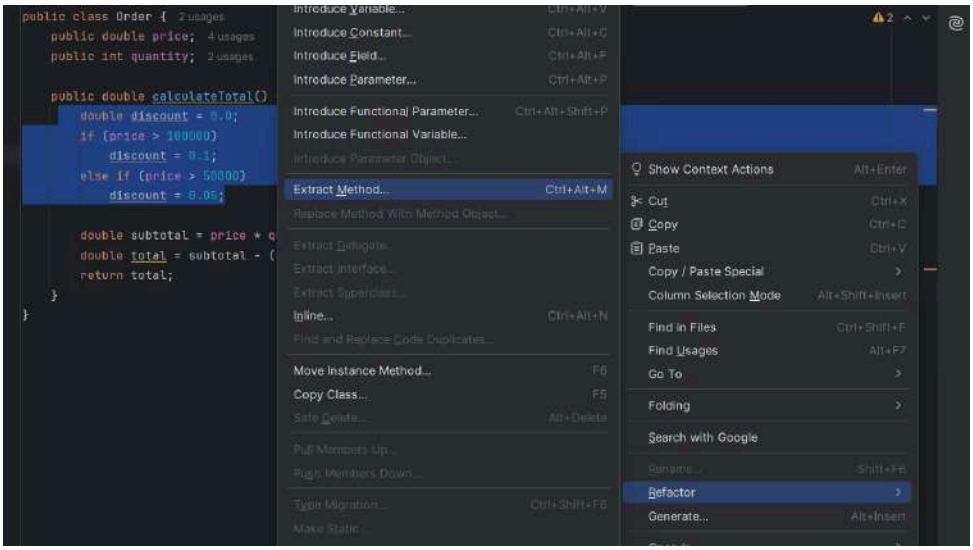
Here's an example of refactoring using the IntelliJ IDEA IDE:

1. The first step is to identify the part of the code you want to refactor. For example, in an order processing program, there might be long logic for calculating discounts and total prices. This code can be extracted into its own method to make it easier to understand.

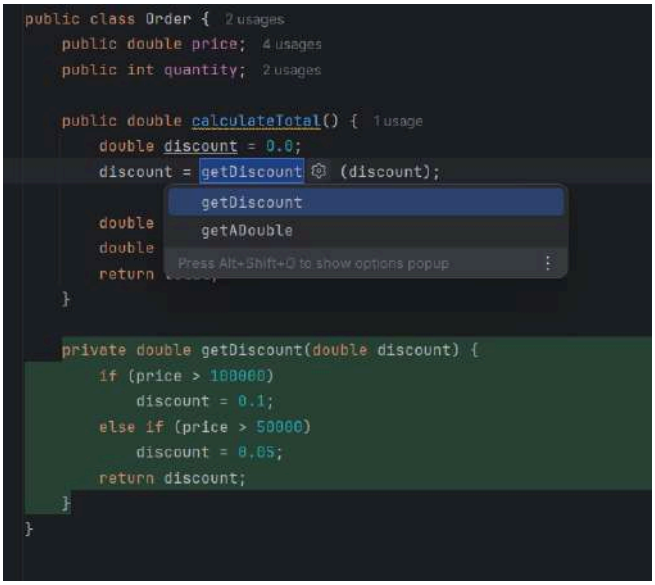
2. Select the lines of code you want to refactor carefully so there are no errors after refactoring. Here, we'll try to do an Extract Method refactor on the code below.

```java
public class Order {  2 usages
    public double price;  4 usages
    public int quantity;  2 usages

    public double calculateTotal() {  1 usage
        double discount = 0.0;
        if (price > 100000)
            discount = 0.1;
        else if (price > 50000)
            discount = 0.05;

        double subtotal = price * quantity;
        double total = subtotal - (subtotal * discount);
        return total;
    }
}
```

3. In the image, there are many refactoring options available. Choose the one that fits your needs. Since we're doing Extract Method, select Extract Method.



4. A dialog will appear to name the new method. A clear name helps readers understand its purpose without opening the implementation. Since the extracted code calculates the discount percentage, you can name it getDiscount().



5. After pressing Enter, the IDE will automatically create a new method from the extracted code. The main code becomes shorter and easier to read, while the calculation logic now lives in a separate method that can be reused in other parts of the program.

REFERENCES

Refactoring - refactoring.guru

RefactoringTechniques - refactoring.guru

RefactoringSourceCode - jetbrains

How to Refactor a Class/Method/Package in IntelliJ Idea? - geeksforgeeks.org

CODELAB

CODELAB 1

```java
// Class Book to store book information
class Book {
    public String title;
    public String author;
    public double price;
    public int stock;

    // Constructor
    Book(String title, String author, double price, int stock) {
        this.title = title;
        this.author = author;
        this.price = price;
        this.stock = stock;
    }

    // Display book details
    public void displayInfo() {
        System.out.println("Title: " + title);
        System.out.println("Author: " + author);
        System.out.println("Price: $" + price);
        System.out.println("Discounted Price $" + (price - (price * 0.1)));
        System.out.println("Stock: " + stock);
    }

    // Adjust the book stock
    public void adjustStock(int adjustment) {
        stock += adjustment;
        System.out.println("Stock adjusted.");
        System.out.println("Current stock: " + stock);
    }
}
```

```java
// Class Library to store library location and a book
class Library {
    public Book book;
    public String location;

    public Library(Book book, String location) {
        this.book = book;
        this.location = location;
    }

    // Display library and book information
    public void showLibraryInfo() {
        System.out.println("Library Location: " + location);
        book.displayInfo();
    }
}
```

```java
class MainApp {
    public static void main(String[] args) {
        Book book1 = new Book("Harry Potter", "J.K. Rowling", 10,2);
        Library lib = new Library(book1, "Perpustakaan Kota");

        // Display initial information
        lib.showLibraryInfo();

        // Add more stock
        book1.adjustStock(5);

        // Display updated information
        lib.showLibraryInfo();
    }
}
```

This program is used to manage books and a library. However, some parts of the code need to be improved to make it clearer and easier to understand. Here is the refactoring plan:

1. In the Book class, add getter and setter methods for the fields title, author, stock, and price. Also, add setters for the book and location fields in the Library class. **(Clue: Encapsulate Field)**

2. Introduce a new constant in the Book class to store the discount value (for example, DISCOUNT_RATE = 0.1). **(Clue: Introduce Constant)**

3. Separate the discount price calculation from displayInfo() into a new method in the Book class named calculateDiscount(). **(Clue: Extract Method)**

   a. Before

```
// Class Book
// Display book details
  public void displayInfo() {
    System.out.println("Title: " + getTitle());
    System.out.println("Author: " + getAuthor());
    System.out.println("Price: $" + getPrice());
    System.out.println("Discounted Price: $" + (getPrice() - (getPrice() * DISCOUNT_RATE)));
    System.out.println("Stock: " + getStock());
  }
```

   b. After

```
// Class Book
// Display book details
public void displayInfo() {
  System.out.println("Title: " + getTitle());
  System.out.println("Author: " + getAuthor());
  System.out.println("Price: $" + getPrice());
  System.out.println("Discounted Price: $" + calculateDiscount());
  System.out.println("Stock: " + getStock());
}
```

4. Move the main() method from the MainApp class to a new class named Main **(create a new class),** and make sure the MainApp class is removed afterward. **(Clue: Move Method)**

TASK

TASK 1

```java
class Doctor {
    private static final double BONUS_RATE = 0.08;
    public String name;
    private int id;
    private double salary;
    private String specialization;

    // Constructor
    public Doctor(String name, int id, double salary, String specialization) {
        this.name = name;
        this.id = id;
        this.salary = salary;
        this.specialization = specialization;
    }

    public void applyBonus(){
        double bonus = salary * BONUS_RATE;
        salary+= bonus;
        System.out.println("Bonus applied ! New Salary : " + salary);
    }

    public void printDetails() {
        System.out.println("Doctor ID: " + id);
        System.out.println("Name: " + name);
        System.out.println("Specialization: " + specialization);
        System.out.println("Salary: $" + salary);
    }

    // Update specialization
    public void updateSpecialization(String newSpecialization) {
        specialization = newSpecialization;
        System.out.println("Specialization updated to: " + specialization);
    }
}

class Patient {
    public String name;
    public int recordNumber;
    public Doctor doctor;
    public String disease;

    // Constructor
    public Patient(String name, int recordNumber, Doctor doctor, String disease) {
        this.name = name;
        this.recordNumber = recordNumber;
        this.doctor = doctor;
        this.disease = disease;
    }

    public void printPatientDetails() {
        System.out.println("Patient Name: " + name);
        System.out.println("Record Number: " + recordNumber);
        System.out.println("Disease: " + disease);
        System.out.println("Doctor in Charge: " + doctor.name);
    }

    public void updateDisease(String newDisease) {
        disease = newDisease;
        System.out.println("Disease updated to: " + disease);
    }
}
```

```java
class Hospital {
    public String hospitalName;
    public String address;
    public Patient patient;

    public Hospital(String hospitalName, String address, Patient patient) {
        this.hospitalName = hospitalName;
        this.address = address;
        this.patient = patient;
    }

    public void printHospitalDetails() {
        System.out.println("Hospital Name: " + hospitalName);
        System.out.println("Address: " + address);
        patient.printPatientDetails();
    }
}

class MainApp {
    public static void main(String[] args) {
        Doctor doctor = new Doctor("Dr. Sarah Lee", 2001, 12000, "Cardiology");
        Patient patient = new Patient("Michael Brown", 555, doctor, "Heart Disease");

        Hospital hospital = new Hospital("City General Hospital", "123 Main Street", patient);
        hospital.printHospitalDetails();

        System.out.println();
        doctor.applyBonus();
        doctor.printDetails();
    }
}
```

To make the code easier to understand, we need to do refactoring. Follow these steps:

1. Create a getter method for the name variable in the Doctor class **(Clue: Encapsulate Field).**

2. Extract the bonus calculation logic into a separate method in the Doctor class named calculateBonus() **(Clue: Extract Method)**.

```java
public void applyBonus(){  1 usage
    double bonus = salary * BONUS_RATE;
    salary+= bonus;
    System.out.println("Bonus applied ! New Salary : " + salary);
}
```

3. Do an Inline Variable refactoring on the bonus variable inside the applyBonus method **(Clue: Inline Variable)**.

4. Create a new class named Main.

5. Move the main method from the MainApp class to the new class named Main using refactoring **(Clue: Move Members)**.

TASK 2

```
public class TaxiTicket {
    public String pName;
    public String sLocation;
    public String dest;
    public double prc;
    private double duration;
    private double speed;

    private static final double MIN_SPEED = 5;
    private static final double MAX_SPEED = 100;

    public TaxiTicket(String passengerName, String startLocation, String destination,
                      double price, double duration, double speed) {
        this.pName = passengerName;
        this.sLocation = startLocation;
        this.dest = destination;
        this.prc = price;
        this.duration = duration;
        this.speed = speed;
    }

    // Method to check taxi status
    public void cS() {
        System.out.println("Your taxi is heading to " + dest);
    }

    // Method to display estimated travel duration
    public void dED() {
        System.out.println("Estimated travel duration: " + duration + " minutes");
    }

    // Method to display the route
    public void dR() {
        System.out.println("Route: " + sLocation + " -> " + dest);
    }

    // Method to slow down the taxi
    public void sLowDown(double speedReduction) {
        speed -= speedReduction;
        if (speed < MIN_SPEED)
            speed = MIN_SPEED;
        duration += speedReduction * 0.5;
        System.out.println("Taxi slowed down! Current speed: " + speed + " km/h");
    }
```

```
    // [ Lanjutan dari kode diatas ]

    // Method to speed up the taxi
    public void speedUp(double speedIncrease) {
        speed += speedIncrease;
        if (speed > MAX_SPEED)
            speed = MAX_SPEED;
        System.out.println("Taxi sped up! Current speed: " + speed + " km/h");
    }

    // Method to display basic info passenger and trip
    public void dI() {
        System.out.println("Passenger Name : " + pName);
        System.out.println("Start Location : " + sLocation);
        System.out.println("Destination    : " + dest);
        System.out.println("Price          : " + prc);
        System.out.println("Final Price    : " + (prc + (prc * 0.1))); // Price including 10% tax
    }

    // Method to display full info including duration and speed
    public void detailedInfo() {
        dI();
        System.out.println("Duration       : " + duration + " minutes");
        System.out.println("Speed          : " + speed + " km/h");
    }

    public static void main(String[] args) {
        TaxiTicket ticket = new TaxiTicket("Alice", "Downtown", "Airport", 50.0, 30.0, 60.0);

        ticket.detailedInfo(); // Display full info

        ticket.cS(); // Check taxi status

        // Display route and estimated duration
        ticket.dR();
        ticket.dED();

        // Simulate slowing down and speeding up
        ticket.slowDown(20);
        ticket.speedUp(15);
    }
}
```

The code above is hard to understand and needs refactoring. Please refactor it following the guidelines below to improve clarity, maintainability, and consistency, while keeping the program functional.

1. Create a new class named MainApp and **use Move Method/Move Members** to move the main() function from the TaxiTicket class to the MainApp class.

2. Use **Rename Method/Variable refactoring** on unclear variable, parameter, and method names. Use the following name changes:

VARIABLE / METHOD NAMES BEFORE REFACTORING	VARIABLE / METHOD NAMES AFTER REFACTORING
public String pName public String sLocation public String desc public double price	public String passengerName public String startLocation public String destination public double price
cS()	checkStatus()
dED()	displayEstimatedDuration()
dR()	displayRoute()
dI()	displayInfo()

3. Add a new constant called TAX_RATE with the value 0.1 inside the **displayInfo()** method.

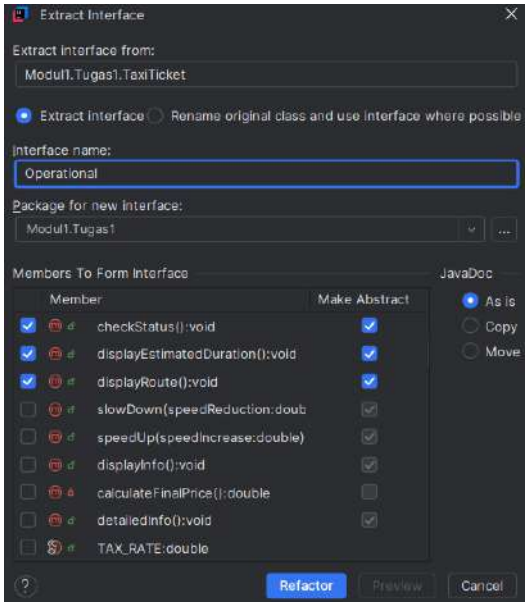
4. Do **Extract Method** on the final price calculation and name the new method **calculateFinalPrice()**.
   a. Before

```
// Method to display basic passenger and trip info
public void displayInfo() {
    System.out.println("Passenger Name : " + passengerName);
    System.out.println("Start Location : " + startLocation);
    System.out.println("Destination    : " + destination);
    System.out.println("Price          : " + price);
    System.out.println("Final Price    : " + (price + (price * TAX_RATE))); // Price including 10% tax
}
```

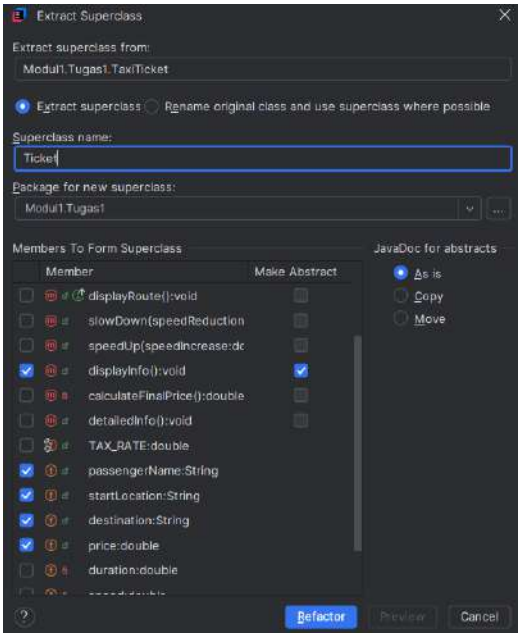
   b. After

```
// Method to display basic passenger and trip info
public void displayInfo() {
    System.out.println("Passenger Name : " + passengerName);
    System.out.println("Start Location : " + startLocation);
    System.out.println("Destination    : " + destination);
    System.out.println("Price          : " + price);
    System.out.println("Final Price    : " + calculateFinalPrice()); // Price including 10% tax
}

private double calculateFinalPrice() {
    return price + (price * TAX_RATE);
}
```

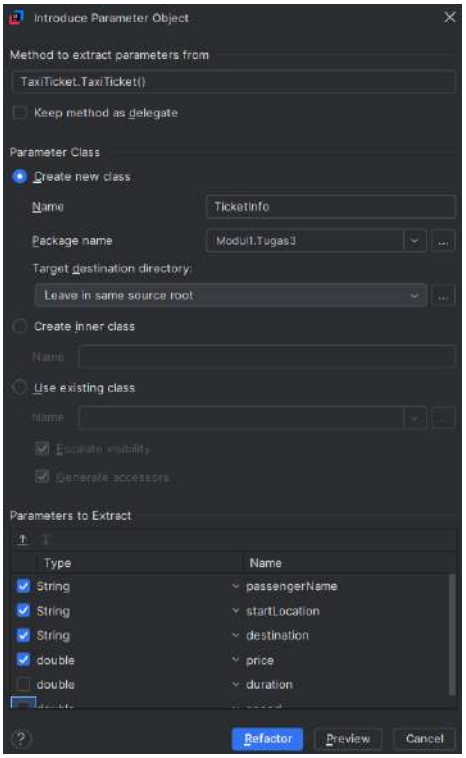
5. **Extract Interface** from the TaxiTicket class to create an **interface named Operational**. Put the methods checkStatus(), displayEstimatedDuration(), and displayRoute() into Operational.



6. **Extract Superclass** from the TaxiTicket class to create a **superclass named Ticket**. Move the attributes passengerName, startLocation, destination, and price, and also select the displayInfo() method and mark it as abstract.



7. **Do Introduce Parameter Object** refactoring in the **TaxiTicket class** for the constructor parameters. Create a new class named TicketInfo and include the attributes passengerName, startLocation, destination, and price.



Note: Do the refactoring live during the demo.

TASK 3

Create a simple program with a theme you chose from the spreadsheet.

Tema Pemrograman Lanjut 2025/2026

In that program, **perform at least 6 refactorings**. Provide **two versions** of the program (before refactoring and after refactoring). Then explain what you refactored in the program.

CRITERIA & ASSESSMENT DETAILS

A. GENERAL ASSESSMENT CRITERIA

ASSESSMENT CRITERIA	POINTS (TOTAL 100%)
CODELAB 1	**Total 15%**
Refactoring based on instructions	10%
Understanding of material	5%
TASK 1	**Total 20%**
Refactoring based on instructions	15%
Understanding of material	5%
TASK 2	**Total 35%**
Refactoring based on instructions	25%
Understanding of material	5%
Code runs without errors	5%
TASK 3	**Total 30%**
Refactoring based on instructions	15%
Understanding of material	10%
Code runs without errors	5%

Note: For detailed point distribution of the **Refactoring based on instructions** in Codelab 1, Task 1, Task 2, and Task 3, please refer to Section **B: DETAILS OF REFACTORING ASSESSMENT.**

B. DETAILS OF REFACTORING ASSESSMENT

DETAILS OF CODELAB 1 ASSESSMENT CRITERIA : REFACTORING BASED ON INSTRUCTIONS (10%)	
Number of Successful Refactorings Performed in Codelab 1	**Score**
0	0
1	40
2	60
3	80
4	100

DETAILS OF TASK 1 ASSESSMENT CRITERIA : REFACTORING BASED ON INSTRUCTIONS (15%)	
Number of Successful Refactorings Performed in Task 1	**Score**
0	0
1	40
2	60
3	80
4	100

DETAILS OF TASK 2 ASSESSMENT CRITERIA : REFACTORING BASED ON INSTRUCTIONS (25%)	
Number of Successful Refactorings Performed in Task 2	**Score**
0	0
1	20
2	40
3	60
4	70
5	80
6	90
7	100

DETAILS OF TASK 3 ASSESSMENT CRITERIA : REFACTORING BASED ON INSTRUCTIONS (15%)	
Number of Successful Refactorings Performed in Task 3	**Score**
0	0
1	30
2	50
3	60
4	70
5	80
6	100