VERSI 0.1

JANUARI 2025

PEMROGRAMAN LANJUT

MODUL 2 - SIMPLE REFACTORING

DISUSUN OLEH:

Ir. Wildan Suharso, M.Kom.

Muhammad Ega Faiz Fadlillah

M. Ramadhan Titan Dwi .C

TIM LABORATORIUM INFORMATIKA

UNIVERSITAS MUHAMMADIYAH MALANG

PENDAHULUAN

TUJUAN

1. Mahasiswa mampu memahamai refactor
2. Mahasiswa mampu memahami teknik-teknik refactor
3. Mahasiswa mamu melakukan/mengimplementasikan refactoring

TARGET MODUL

1. Mahasiswa dapat memahami konsep refactoring
2. Mahasiswa mampu mengimplementasikan konsep refactoring

PERSIAPAN

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

KEYWORDS

Refactoring

TABLE OF CONTENTS

TEORI

REFACTORING

A. Apa itu Refactoring

Refactoring kode adalah proses memperbaiki struktur internal dari kode sumber tanpa mengubah cara kerjanya. Dengan kata lain, refactoring tidak menambahkan fitur baru, tetapi fokus pada penyempurnaan kode agar lebih rapi, efisien, dan mudah untuk dirawat.

Tujuan utama dari refactoring adalah untuk meningkatkan kualitas kode. Kode yang bersih dan terorganisir membantu pengembang bekerja dengan lebih efisien, mengurangi kemungkinan terjadinya kesalahan, serta mempermudah pemeliharaan di masa depan.

B. Kapan Perlu Dilakukan Refactoring

- **Sebelum Menambahkan Fitur Baru**

  Melakukan refactoring sebelum menambahkan fitur baru sangat penting untuk memastikan bahwa struktur kode yang ada sudah dalam kondisi optimal. Dengan mempersiapkan kode yang bersih dan terorganisir, dapat mengintegrasikan fitur baru dengan lebih mudah dan efisien.

- **Ketika Mengatasi Bug**

  Refactoring saat memperbaiki bug membantu menemukan akar masalah lebih cepat. Dengan struktur kode yang lebih rapi, proses perbaikan menjadi lebih mudah dan mengurangi risiko munculnya bug baru.

- **Menghadapi Code Smells**

  Code smells adalah istilah yang digunakan untuk menggambarkan indikasi adanya masalah dalam kode, seperti metode yang terlalu panjang, kode yang duplikat, atau penggunaan nama variabel yang tidak jelas. Dengan melakukan refactoring, dapat menghilangkan ketidakjelasan dan kompleksitas yang tidak perlu, sehingga kode menjadi lebih mudah dipahami dan dikelola.

- **Melalui Proses Code Review**

  Code review adalah proses evaluasi di mana kode yang ditulis oleh satu pengembang diperiksa oleh pengembang lain atau tim. Proses ini bertujuan untuk memastikan bahwa kode memenuhi standar kualitas tertentu.

C. Mengapa Perlu Melakukan Refactoring

Refactoring adalah proses untuk menyederhanakan dan memperbaiki struktur kode tanpa mengubah fungsionalitasnya. Tujuannya adalah agar kode menjadi lebih mudah dipahami dan lebih ringkas. Berikut adalah beberapa alasan mengapa refactoring sangat diperlukan :

- **Meningkatkan Pemahaman Kode** : Refactoring membuat kode lebih jelas, sehingga pengembang lain dapat lebih mudah memahami apa yang dilakukan oleh kode tersebut.
- **Membantu Menemukan dan Memperbaiki Bug** : Dengan menyederhanakan kode, refactoring memudahkan pengembang untuk menemukan dan memperbaiki kesalahan yang mungkin ada.
- **Mempercepat Proses Pengembangan** : Kode yang lebih terorganisir dan sederhana memungkinkan pengembang untuk bekerja lebih cepat, sehingga proses pengembangan perangkat lunak menjadi lebih efisien.
- **Meningkatkan Desain Perangkat Lunak** : Refactoring membantu meningkatkan kualitas desain perangkat lunak secara keseluruhan, membuatnya lebih mudah untuk dikelola dan dikembangkan di masa depan.

D. Teknik Refactoring

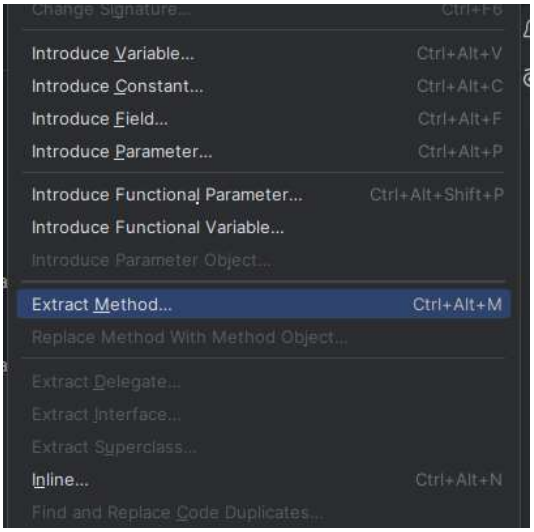
Berikut ini adalah beberapa teknik refactoring yang umum digunakan :

1. Extract Method

Extract Method digunakan untuk memecah method yang panjang dan kompleks menjadi bagian-bagian yang lebih kecil. Teknik ini dilakukan dengan memindahkan bagian kode tertentu ke dalam sebuah metode baru, sehingga kode utama menjadi lebih ringkas, terstruktur, dan mudah dipahami.

Cara Melakukan Extract Method :

- Blok bagian kode yang ingin dipisahkan
- Klik kanan pada kode tersebut, pilih **Refactor → Extract Method**
- Berikan nama metode sesuai dengan fungsi atau tujuan dari kode tersebut



a. Before

Pada contoh kode dibawah, terdapat struktur if-else di mana pilihan 1 digunakan untuk menambahkan item. Namun, proses penambahan item dituliskan langsung di dalam blok case, sehingga kode menjadi panjang dan kurang terstruktur.

```
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

Setelah dilakukan Extract Method, blok kode yang menangani input detail item seperti **product name, item price** dan **item quantity** akan direfactor ke dalam metode baru bernama **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);
    }
```

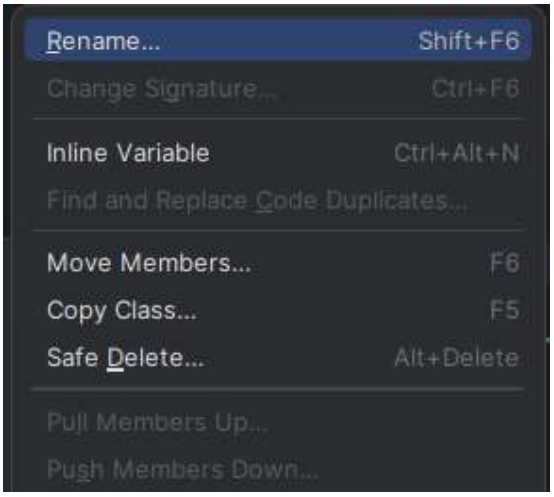
```
private static void inputItemDetails(Scanner scanner, ArrayList<String> itemNames, String name,   1 usage
                                     ArrayList<Double> itemPrices, ArrayList<Integer> itemQuantities) {
    System.out.print("Enter item price: ");
    double price = Double.parseDouble(scanner.nextLine());

    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

Teknik Rename Method/Variable adalah proses mengganti nama method atau variabel yang sudah ada dengan nama yang lebih tepat, serta secara otomatis memperbarui semua referensi terkait di seluruh kode. Dengan demikian, kita tidak perlu melakukan perubahan secara manual satu per satu pada setiap bagian kode yang menggunakannya.

Cara Melakukan Rename Method/Variable :

- Pilih variabel atau method yang ingin diubah namanya.
- Klik kanan pada variabel atau method tersebut → pilih **Refactor** → **Rename**
- Masukkan nama baru untuk method/variable.



a. Before

Pada contoh di bawah terdapat variabel bernama **result** yang digunakan untuk menyimpan nilai gaji setelah dipotong pajak. Namun, penamaan variabel tersebut kurang deskriptif sehingga dapat membingungkan pembaca dalam memahami fungsinya.

```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

Setelah dilakukan refactoring, nama variabel result diubah menjadi **salaryAfterTax**, yang merupakan penamaan lebih deskriptif.

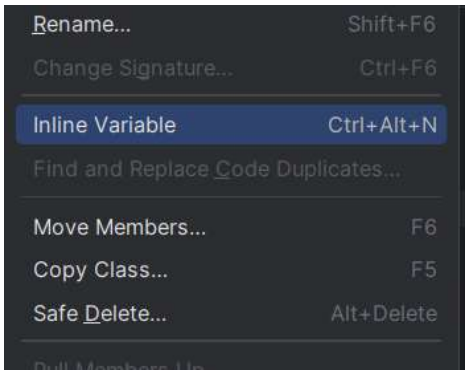
```
public class Salary {
    public static void main(String[] args) {
        double baseSalary = 3000.0;
        double taxRate = 0.15;

        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 digunakan untuk mengganti variabel yang nilainya sederhana dan hanya digunakan satu kali, dengan langsung menuliskan nilai atau ekspresi tersebut pada tempat penggunaannya. Tujuan utamanya adalah untuk membuat kode lebih ringkas dan mudah dibaca, serta mengurangi kompleksitas akibat penggunaan variabel yang tidak terlalu dibutuhkan.

Cara Melakukan Inline Variable

- Pilih variabel yang hanya digunakan satu kali dan memiliki nilai yang sederhana.
- Klik kanan pada variabel → pilih **Refactor** → lalu **Inline Variable.**



a. Before

Pada contoh di bawah, variabel **area** hanya digunakan satu kali dan berfungsi menyimpan nilai sederhana berupa perhitungan luas lingkaran. Karena penggunaannya hanya sekali, kita bisa menyederhanakannya.

```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

Setelah dilakukan refactoring, variabel area yang berisi ekspresi **3.14 * radius * radius** dihapus dari kode. Ekspresi tersebut akan diletakkan langsung di bagian kita menggunakan variabel area sebelumnya atau di dalam fungsi **System.out.println()**.

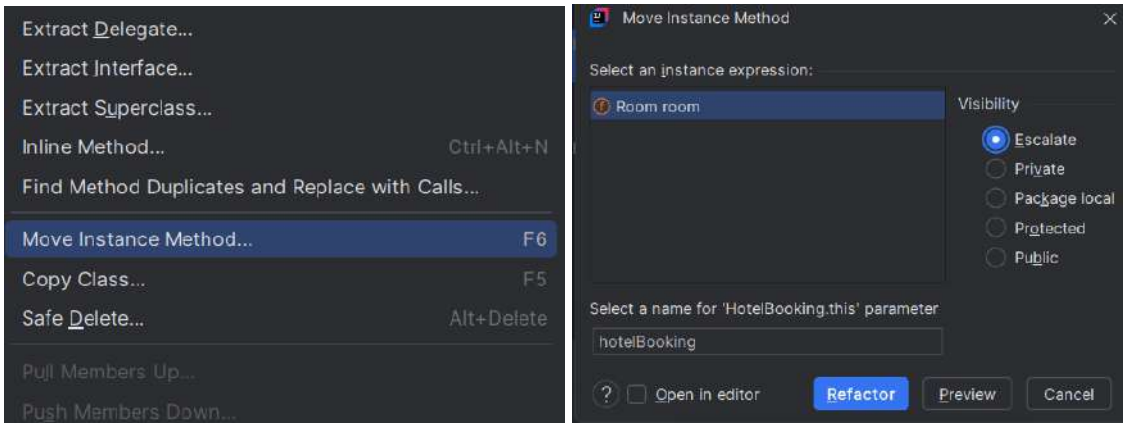
```java
import java.util.Scanner;

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();

        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 atau Move Field adalah teknik refactoring di mana sebuah metode atau atribut dipindahkan dari satu kelas ke kelas lain yang lebih relevan atau lebih sesuai dengan fungsinya. Tujuannya adalah agar kode menjadi lebih terstruktur, modular, dan mudah dipelihara.

Cara Melakukan Move Method / Field

- Pilih salah satu method yang ingin dipindah ke class lain
- Klik kanan pilih → pilih **Refactor** → lalu **Move Instance Method**.
- Tentukan kelas tujuan tempat metode atau atribut tersebut akan dipindahkan.
- Jika memindahkan method, berikan nama baru untuk objek atau metode tersebut jika diperlukan



a. Before

Pada awalnya, class HotelBooking memiliki method **calculateTotalPrice()** dan **calculateFinalPrice()**, tetapi penempatan kedua metode tersebut kurang sesuai karena logikanya lebih terkait langsung dengan properti kamar, seperti harga per malam

```
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

Setelah melakukan refactoring, kita memindahkan kedua method tersebut ke dalam class Room, sehingga method tersebut sudah berada di class yang lebih relevan yaitu berhubungan dengan kamar.

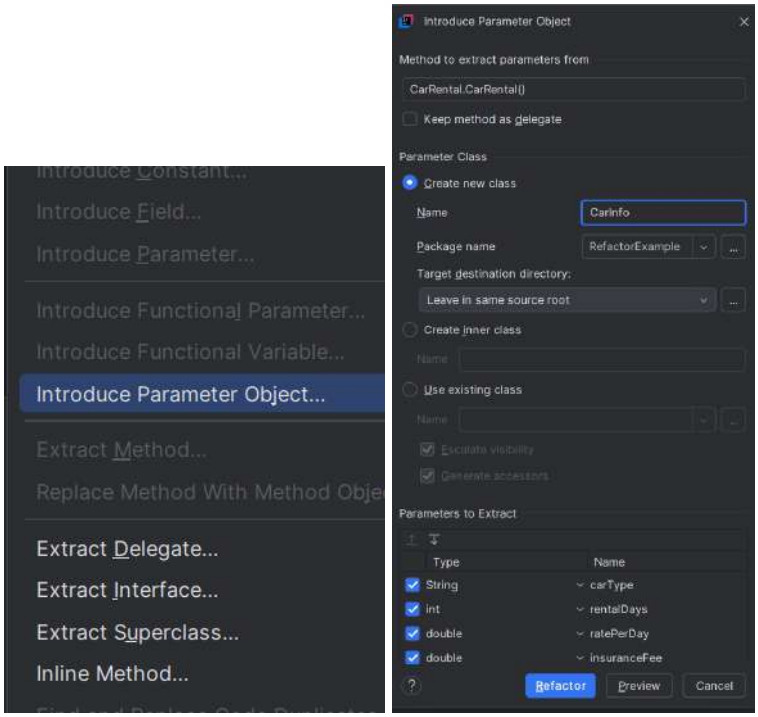
```
class Room {
    private String roomType;
    private double pricePerNight;

    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

Teknik ini digunakan untuk menyederhanakan method yang memiliki sekelompok parameter yang sering muncul bersama. Daripada terus-menerus menuliskan beberapa parameter yang serupa di method, kita bisa menggabungkan parameter - parameter tersebut ke dalam sebuah kelas khusus (object). Dengan cara ini, metode yang sebelumnya memiliki banyak parameter kini cukup menerima satu objek saja sebagai parameter.

Cara Melakukan Introduce Parameter Object

- Pilih salah satu method yang parameternya ingin dijadikan sebuah objek
- Klik kanan → pilih **Refactor** → lalu **Introduce Parameter Object**.
- Beri nama class baru dan pilih parameter apa saja yang ingin di refactor



a. Before

Pada method **CarRental()** dan **calculateTotalCost()** memiliki parameter yang serupa yaitu **int rentalDays**, **double ratePerDay** dan **double insuranceFee**. Disini kita akan melakukan refactoring introduce parameter object.

```java
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

Setelah melakukan introduce paramter object maka akan di buatkan sebuah record class baru yaitu **CarInfo** yang berisi parameter **int rentalDays, double ratePerDay dan double insuranceFee**. Record class dirancang untuk menyederhanakan pembuatan class yang tujuannya hanya menyimpan data (data carrier / data holder).

```java
package RefactorExample;

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

Pada constructor **CarRental()** yang awalnya berisi parameter int rentalDays, double ratePerDay dan double insuranceFee maka akan berubah menjadi **CarInfo carInfo** akibat refactor introduce parameter object.

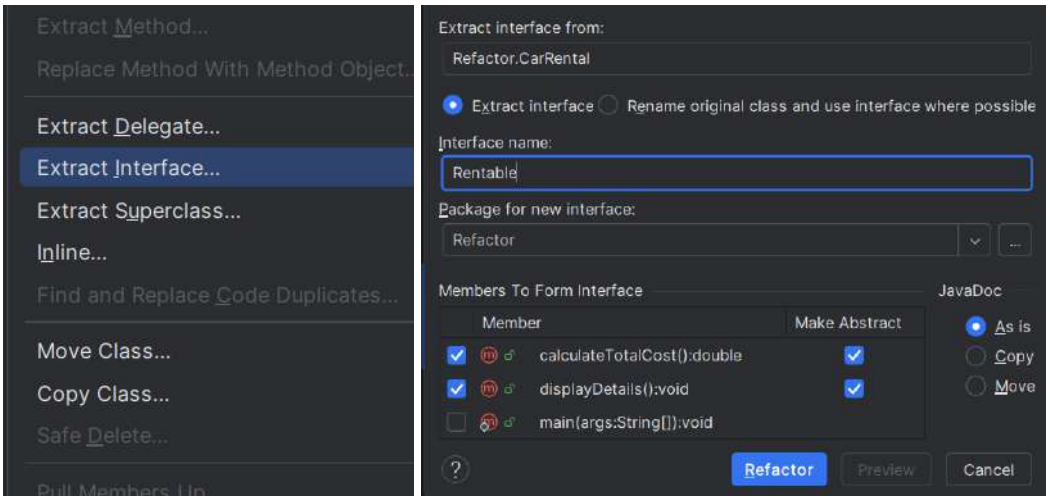
```java
public class CarRental {
    private final String carType;
    private final int rentalDays;
    private final double ratePerDay;
    private final double insuranceFee;

    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 adalah teknik dalam pemrograman yang digunakan untuk membuat interface dari kelas yang ada. Selain itu teknik ini memfasilitasi abstraksi dan mempermudah penggunaan variasi kelas yang berbeda.

Cara Melakukan Extract Interface

- Pilih salah satu method yang parameternya ingin dijadikan sebuah objek
- Klik kanan → pilih **Refactor** → lalu **Extract Interface**
- Beri nama class interface dan pilih parameter apa saja yang ingin di refactor



a. Before

Pada class CarRental terdapat method seperti **calculateTotalCost()** dan **displayDetails()**. Namun, class ini belum memiliki abstraksi, sehingga kita tidak bisa dengan mudah membuat variasi jenis rental lain (misalnya LuxuryCarRental, ElectricCarRental, atau MotorcycleRental) tanpa harus mengubah struktur class yang ada.

```java
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

Setelah dilakukan extract interface maka akan dibuatkan interface dengan nama **Rentable**. Interface ini berisi method **calculateTotalCost()** dan **displayDetails()**.

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

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

Class CarRental akan mengimplementasikan interface Rentable secara langsung dan 2 method sebelummnya yaitu **calculateTotalCost()** dan **displayDetails()** akan dioverride akibat implementasi interface Rentable

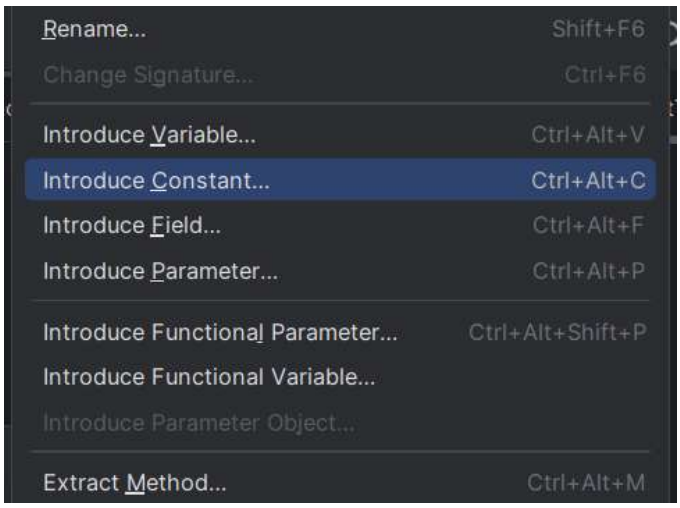
```java
public class CarRental implements Rentable {
    private final String carType;  2 usages
    private final int rentalDays;  3 usages
    private final double ratePerDay;  3 usages
    private final double insuranceFee;  3 usages

    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

Teknik ini digunakan untuk mengganti angka atau nilai tetap (magic number) yang langsung ditulis di dalam kode dengan sebuah konstanta baru bernama (symbolic constant). Tujuannya agar kode lebih mudah dipahami, karena nama konstanta dapat menjelaskan maksud dari nilai tersebut.

Cara Melakukan Replace Magic Number with Symbolic Constant

- Pilih salah satu nilai atau value yang ingin dijadian sebuah konstanta
- Klik kanan → pilih **Refactor** → lalu **Introduce Constant**
- Beri nama konstanta



a. Before

Di method **CalculateTotalCost()** terdapat nilai 100000 dan 0.1. Angka tersebut disebut sebagai magic number karena nilainya ditulis begitu saja. Hal ini membuat pembaca kode sulit memahami bahwa 100000 sebenarnya batas minimal belanja untuk mendapat diskon, dan 0.1 adalah persentase diskonnya. Supaya lebih mudah dipahami, sebaiknya angka tersebut diganti dengan konstanta yang memiliki nama bermakna, misalnya **DISCOUNT_THRESHOLD** dan **DISCOUNT_RATE**.

```java
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

Setelah melakukan refactoring, akan terbuat variabel constant yaitu **int DISCOUNT_THRESHOLD** dengan nilai 100000 dan **double DISCOUNT_RATE** dengan nilai 0.1.

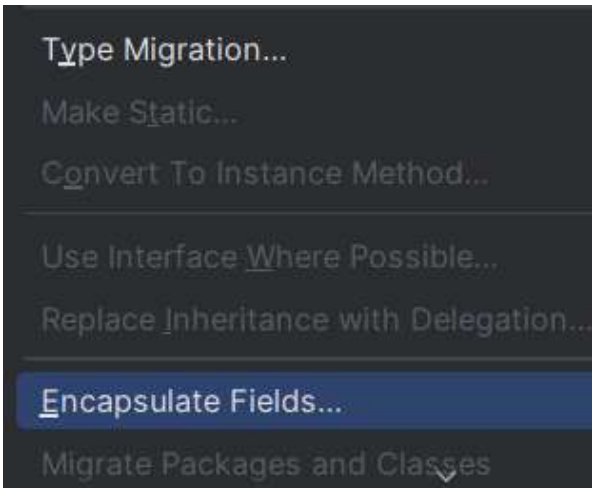
```java
public class ShopTransaction {

    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;
    }
```

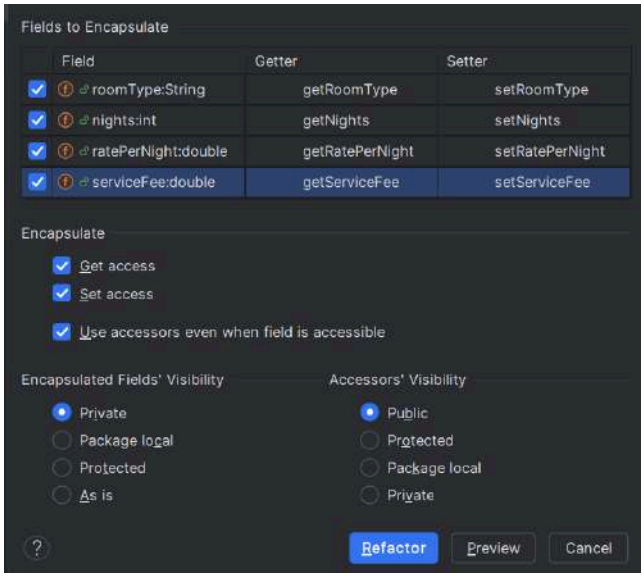
8. Encapsulate Field

Encapsulation atau pembungkusan data merupakan prinsip penting dalam pemrograman berorientasi objek yang bertujuan untuk melindungi atribut di dalam sebuah kelas agar tidak bisa diakses atau diubah secara langsung dari luar. Teknik ini biasanya dilakukan dengan menjadikan atribut bersifat private lalu menyediakan metode getter untuk membaca nilainya dan setter untuk mengubahnya.

Cara Melakukan Encapsulate Field

- Pilih salah satu atribut yang ingin di encapsulate field
- Klik kanan → pilih **Refactor** → lalu **Introduce Constant**
- Pilih atribut apa saja yang ingin di encapsulate





a. Before

Pada kelas HotelBooking memiliki empat atribut yaitu roomType, nights, ratePerNight, dan serviceFee yang semuanya menggunakan visibility public. Hal ini membuat atribut tersebut dapat diakses dan dimodifikasi secara langsung dari luar kelas tanpa adanya kontrol, sehingga berpotensi menimbulkan masalah keamanan maupun inkonsistensi data.

```java
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

Setelah dilakukan refactoring maka keempat atribut yang sebelumnya memiliki visibility public akan diubah menjadi private. Sebagai gantinya, disediakan metode getter untuk mengambil nilai atribut dan setter untuk mengubah nilainya.

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

    public String getRoomType() {
        return roomType;
    }

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

    public int getNights() {
        return nights;
    }

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

    public double getRatePerNight() {
        return ratePerNight;
    }

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

    public double getServiceFee() {
```

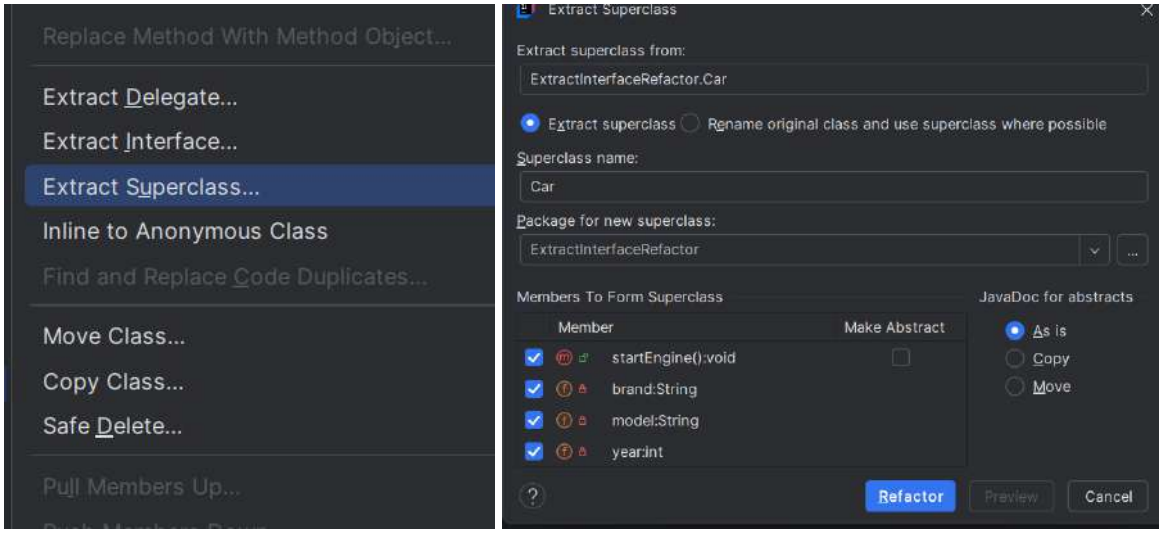
9. Extract Superclass

Extract Superclass adalah salah satu teknik di mana kita membuat sebuah kelas induk (superclass) baru untuk menampung atribut dan method yang dimiliki bersama oleh dua atau lebih kelas. Setelah itu, kelas-kelas akan dijadikan subclass dan mewarisi (inherit) dari superclass tersebut. Tujuannya adalah untuk mengurangi duplikasi kode dan membuat struktur program lebih teratur.

Semisal, ada dua kelas Car dan Motorcycle yang sama-sama punya atribut **brand, model**, dan **year**, daripada menduplikasi atribut tersebut di masing-masing kelas, kita bisa melakukan extract superclass ke kelas Vehicle. Dengan begitu, Car dan Motorcycle cukup fokus pada logika khusus mereka (misalnya openTrunk() untuk mobil atau popWheelie() untuk motor), sementara data umum seperti informasi merek, model, dan tahun disimpan di superclass.

Cara Melakukan Extract Superclass

- Pilih salah satu kelas yang memiliki atribut atau method umum yang juga dimiliki oleh kelas lain.
- Klik kanan → pilih **Refactor** → lalu **Extract Superclass**
- Beri nama untuk superclass
- Pilih atribut dan method mana saja yang ingin dipindahkan ke superclass agar tidak terjadi duplikasi kode. Selain itu, kita juga memiliki opsi untuk menjadikan method tertentu sebagai **abstract method** di superclass



a. Before

Pada kondisi awal, terdapat dua kelas yaitu Car dan Motorcycle. Kedua kelas ini memiliki kesamaan dari sisi atribut maupun method. Dari sisi atribut, keduanya sama-sama menyimpan data tentang **brand**, **model**, dan **year** sedangkan dari sisi method yaitu **startEngine().** Dengan adanya duplikasi atribut dan method seperti ini, kode menjadi kurang efisien dan berpotensi menyulitkan pemeliharaan jika nantinya ada perubahan.

```
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

Setelah dilakukan refactoring dengan extract superclass, dibuatlah sebuah kelas baru bernama **Vehicle** yang berfungsi sebagai superclass untuk menampung atribut dan method yang sama dari **Car** dan **Motorcycle**. Atribut umum seperti **brand, model**, dan **year** sekarang diletakkan di dalam **Class Vehicle**, begitu juga dengan method **startEngine()** yang sudah tidak perlu lagi ditulis ulang di masing-masing kelas turunan. Dengan begitu, **Class Car** hanya fokus pada atribut dan perilaku khusus miliknya, misalnya **trunkCapacity** dan method **openTrunk()**, sementara **Class Motorcycle** tetap punya atribut tambahan **hasSidecar** dan method khusus **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);
    }
}
```

```
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");
    }
}
```

```
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. Bagaimana Cara Melakukan Refactoring?

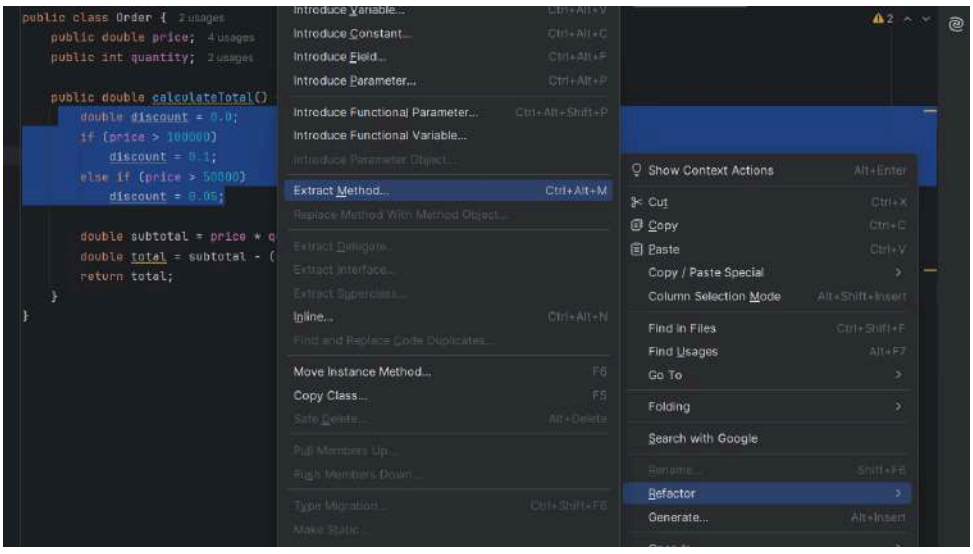
Berikut adalah contoh refactoring dengan menggunakan IDE Intellij IDEA :

1. Langkah pertama adalah menentukan bagian kode yang ingin dipisahkan menjadi direfactor. Misalnya, dalam program pemesanan barang, kita memiliki logika perhitungan diskon dan total harga yang panjang. Kode ini bisa diekstrak menjadi metode tersendiri agar lebih mudah dipahami.

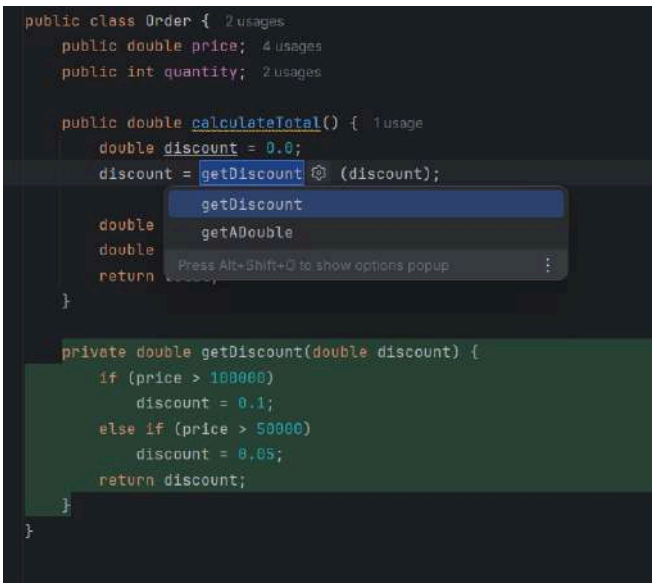
2. Pilih baris kode yang ingin direfactor dengan benar-benar agar setelah di refactor tidak ada kesalahan. Disini kita akan coba melakukan refactor extract method kode dibawah.

```
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. Pada gambar dibawah, banyak pilihan untuk refactor yang dapat digunakan. Pilih yang sesuai dengan kebutuhan kita. Karena kita akan melakukan extract method, maka pilih extract method.



4. Nantinya akan muncul menu untuk memberi nama metode baru. Nama yang jelas akan memudahkan pembaca kode untuk memahami fungsinya tanpa harus membuka implementasi metode. Karena kode yang diekstrak berfungsi untuk mencari persentase diskon maka dapat diberi nama **getDiscount().**



5. Setelah menekan Enter, IDE akan secara otomatis membuat metode baru berdasarkan kode yang diekstrak. Kode utama menjadi lebih ringkas dan mudah dibaca, sedangkan logika perhitungan kini berada di metode tersendiri yang dapat dipanggil kembali di bagian lain program.

REFERENSI

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();
    }
}
```

Program ini digunakan untuk mengelola data buku dan perpustakaan. Namun, ada beberapa bagian kode yang masih perlu diperbaiki supaya lebih jelas dan mudah dipahami. Berikut rencana refactoring yang akan dilakukan :

1. Pada class Book, tambahkan setter dan getter untuk field title, author, stock, dan price. Selain itu, buat juga setter untuk field book dan location pada Class Library. (**Clue: Encapsulate Field**)

2. Perkenalkan sebuah konstanta baru di Class Book untuk menyimpan nilai diskon (misalnya DISCOUNT_RATE = 0.1). (Clue: **Introduce Constant**)

3. Pisahkan perhitungan harga diskon dari displayInfo() menjadi sebuah metode baru di kelas Book dengan nama 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. Pindahkan method main() dari class MainApp ke dalam kelas baru bernama Main **(buat baru)** dan pastikan bahwa kelas MainApp dihapus setelahnya. (**Clue: Move Method**)

TUGAS

TUGAS 1

```
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();
    }
}
```

Agar lebih mudah dipahami, kode di atas perlu dilakukan refactoring. Lakukan refactoring sesuai instruksi berikut:

1. Buatlah method getter dari variable name dalam kelas Doctor (**Clue: Encapsulate Field**)

2. Ekstrak logika perhitungan bonus ke dalam method terpisah dalam kelas Doctor dengan nama 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. Lakukan refactoring Inline Variable pada variable bonus ke dalam method applyBonus (**Clue : Inline Variable**)

4. Buatlah Class baru dengan nama Main

5. Pindahkan method main dari class MainApp ke dalam Class baru bernama Main menggunakan teknik refactoring (**Clue** : **Move Members**)

TUGAS 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);
    }
}
```

Penulisan kode diatas masih sulit untuk dipahami sehingga perlu dilakukan refactoring. Lakukanlah refactoring sesuai dengan petunjuk dibawah ini.

1. Buatlah sebuah class baru bernama MainApp dan lakukan **Move Method/Move Members** dengan memindahkan fungsi main() dari class TaxiTicket menuju class MainApp

2. Lakukan teknik refactoring **Rename Method /Variable** pada penulisan variabel, paramter dan method yang sulit dipahami. Berikut adalah table penamaan yang benar.

PENAMAAN VARIABEL / METHOD SEBELUM REFACTORING	PENAMAAN VARIABEL / METHOD SESUDAH 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. Perkenalkan sebuah constanta baru pada method displayInfo() yaitu 0.1 dengan nama **TAX_RATE**

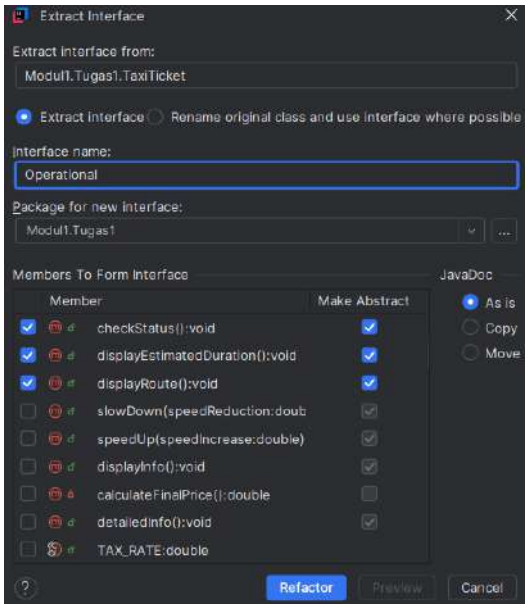
4. Lakukan refactoring **Extract Method** pada bagian perhitungan final price dengan nama method baru yaitu calculateFinalPrice()
   a. Before

```
// 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
}
```

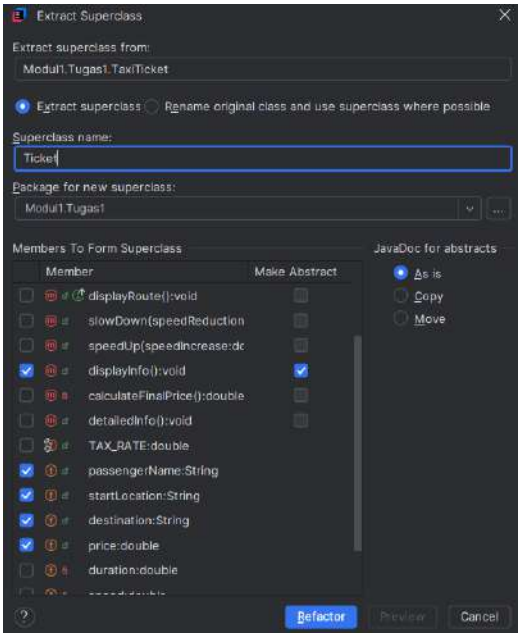
   b. After

```
// 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);
}
```

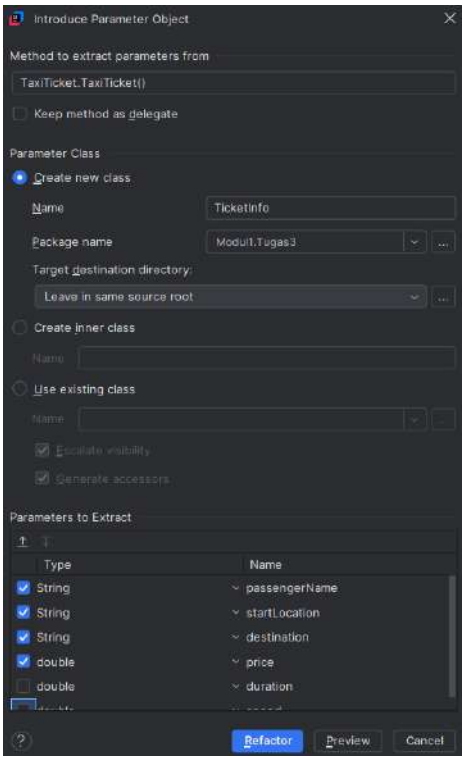
5. **Extract Interface** dari kelas TaxiTicket sehingga menjadi interface dengan nama **Operational**. Method yang akan diletakkan di dalam interface **Operational** yaitu checkStatus(), displayEstimatedDuration() dan displayRoute().



6. **Extract Superclass** dari kelas TaxiTicket sehingga menjadi super class dengan nama **Ticket**. Lalu pilih atribut passengerName, startLocation, destination, price serta pilih juga method displayInfo() dan pastikan centang abstract.



7. Lakukanlah refactoring **Introduce Parameter Object** pada class TaxiTicket bagian parameter dari Constructor Method TaxiTicket dengan nama **TicketInfo** dan centang atribut passangerName, startLocation, destination dan price.



CATATAN : REFACTOR DILAKUKAN SECARA LIVE SAAT DEMO

TUGAS 3

Buatlah sebuah program sederhana dengan tema yang telah kalian tentukan pada Spreadsheet ini :

Tema Pemrograman Lanjut 2025/2026

Pada program tersebut lakukan **minimal 6 refactoring**, buatlah menjadi **dua program (sebelum refactoring dan setelah refactoring).** Lalu jelaskan apa saja yang di refactoring pada program tersebut.

KRITERIA & DETAIL PENILAIAN

A. *KRITERIA PENILAIAN UMUM*

KRITERIA PENILAIAN	POIN (TOTAL 100%)
CODELAB 1	**Total 15%**
Refactoring sesuai instruksi	10%
Pemahaman Materi	5%
TUGAS 1	**Total 20%**
Refactoring sesuai instruksi	15%
Pemahaman Materi	5%
TUGAS 2	**Total 35%**
Refactoring sesuai instruksi	25%
Pemahaman Materi	5%
Kode berjalan tanpa error	5%
TUGAS 3	**Total 30%**
Refactoring sesuai instruksi	15%
Pemahaman Materi	10%
Kode berjalan tanpa error	5%

Catatan : Untuk detail pembagian poin penilaian **Refactoring Sesuai Instruksi** pada Codelab 1, Tugas 1, Tugas 2, dan Tugas 3, dapat dilihat pada bagian **B: DETAIL PENILAIAN REFACTORING**

B. DETAIL PENILAIAN REFACTORING

DETAIL PENILAIAN KRITERIA CODELAB 1 : REFACTORING SESUAI INSTRUKSI (10%)	
Jumlah Refactoring yang Berhasil Dilakukan pada Codelab 1	**Nilai**
0	0
1	40
2	60
3	80
4	100

DETAIL PENILAIAN KRITERIA TUGAS 1 : REFACTORING SESUAI INSTRUKSI (15%)	
Jumlah Refactoring yang Berhasil Dilakukan pada Tugas 1	**Nilai**
0	0
1	40
2	60
3	80
4	100

DETAIL PENILAIAN KRITERIA TUGAS 2 : REFACTORING SESUAI INSTRUKSI (25%)	
Jumlah Refactoring yang Berhasil Dilakukan pada Tugas 2	**Nilai**
0	0
1	20
2	40
3	60
4	70
5	80
6	90
7	100

DETAIL PENILAIAN KRITERIA TUGAS 3 : REFACTORING SESUAI INSTRUKSI (15%)	
Jumlah Refactoring yang Berhasil Dilakukan pada Tugas 3	**Nilai**
0	0
1	30
2	50
3	60
4	70
5	80
6	100