

A. Informasi Praktikan & Ringkasan Pemahaman**1. Identitas**

NIM	11S23024
Nama	Glen Rejeki Sitorus
Kelas	12IF1
Judul Praktikum	Java OOP
Video Presentasi	https://youtu.be/DhHtAWNcDHc

2. Capaian & Ringkasan Pemahaman

Hasil Capaian setelah mengikuti praktikum:

- A (Penyelesaian): Selesai (1), Tidak Selesai (0).
- B (Pemahaman): Tidak Paham (1), Kurang Paham (2), Cukup Paham (3), Paham (4), Sangat Paham (5).

No	Kriteria	A	B
1	Latihan Java OOP	1	3
2	Mengimplementasikan Class Diagram	1	5
3	Studi Kasus 1: Student Information	1	3

Pemahaman yang saya dapat setelah menyelesaikan praktikum adalah bagaimana cara agar dapat membuat code sesuai dengan yang diinginkan, supaya outputnya dapat diterima di delcom.

3. Foto Praktikan



B. Penilaian Observasi

Setiap Mahasiswa yang mengikuti kegiatan praktikum perlu melakukan penilaian terhadap diri sendiri dan rekan kerja berdasarkan [[Indikator Penilaian Observasi Praktikum](#)].

1. Penilaian Terhadap Diri Sendiri

No	Indikator	Skala (1 - 5)
1	Kepatuhan Terhadap Prosedur	3
2	Kemampuan Penyelesaian Tugas	3
3	Kerja Sama Tim	5
4	Inisiatif dan Problem Solving	3
5	Kedisiplinan dan Etika Kerja	3

2. Penilaian Terhadap Rekan Kerja

Harus melakukan penilaian terhadap rekan kerja selama aktivitas praktikum berlangsung.

a) Daniel L. Tobing (11S23025)

No	Indikator	Skala (1 - 5)
1	Kepatuhan Terhadap Prosedur	3
2	Kemampuan Penyelesaian Tugas	3
3	Kerja Sama Tim	5
4	Inisiatif dan Problem Solving	3
5	Kedisiplinan dan Etika Kerja	3

b) Rut Angelika Manurung (11S23027)

No	Indikator	Skala (1 - 5)
1	Kepatuhan Terhadap Prosedur	3
2	Kemampuan Penyelesaian Tugas	3
3	Kerja Sama Tim	5
4	Inisiatif dan Problem Solving	3
5	Kedisiplinan dan Etika Kerja	3

C. Laporan Aktivitas Praktikum

1. Latihan Java OOP

a. Clas

```
Person.java > Person
1  class Person {
2      ^
3  }
```

b. Object

```
public class App {
    Run | Debug
    public static void main(String[] args) {
        var person1 = new Person();
        Person person2 = new Person();
        Person person3;
        person3 = new Person();
        System.out.println(person1);
        System.out.println(person2);
        System.out.println(person3);
    }
}
```

Analisis kegunaan dari kode program pada line 4, 5, 6, dan 7. Amati hasilnya pada terminal.

- Di sini, person1 dideklarasikan menggunakan var, yang secara otomatis mengidentifikasi tipe data sebagai Person. Ini menciptakan objek baru dari kelas Person dan menyimpannya di variabel person1.
- Pada baris ini, person2 secara eksplisit dideklarasikan sebagai tipe Person dan juga diinisialisasi dengan objek baru dari kelas Person. Ini setara dengan baris sebelumnya tetapi menggunakan deklarasi tipe yang lebih jelas.
- Di sini, person3 dideklarasikan tetapi belum diinisialisasi. Ini berarti person3 adalah variabel yang dapat menampung objek Person, tetapi saat ini tidak mengacu pada objek apapun.
- Pada baris ini, person3 diinisialisasi dengan objek baru dari kelas Person. Sekarang person3 juga merujuk pada objek Person yang valid.

c. Field

```
public class App {
    public static void main(String[] args)
        person1.name = "Abdullah";
        person1.address = "Sitoluama";
        // person1.country = "Singapura";

        Person person2 = new Person();
        💡 Person person3;
        person3 = new Person();
        System.out.println(person1);
        System.out.println(person2);
        System.out.println(person3);
    }
}
```

Analisis kegunaan kode program pada line 5, 6, 9, 10, dan 11. Mengapa kode program pada line 7, jika dihilangkan komen-nya akan mengalami error.

- **Kegunaan:** Membuat objek person1 dari kelas Person. Ini memungkinkan Anda untuk mengakses dan mengatur atribut kelas Person.
- **Kegunaan:** Mengatur atribut name dari objek person1 menjadi "Abdullah". Ini menyimpan informasi nama untuk objek tersebut.
- **Kegunaan:** Membuat objek person2 baru dari kelas Person. Ini adalah instansiasi lain yang memungkinkan Anda untuk memiliki objek yang terpisah dari person1.
- **Kegunaan:** Mendeklarasikan variabel person3 dari tipe Person. Namun, saat ini person3 belum diinisialisasi dengan objek apa pun.
- Jika Anda menghapus komentar pada baris ini, Anda akan mengalami error. Penyebabnya adalah bahwa country dideklarasikan sebagai final dalam kelas Person, yang berarti nilainya tidak dapat diubah setelah diinisialisasi.

d. Method

```
class Person {
    String name;
    String address;
    final String country = "Indonesia";
    void sayHello(String paramName) {
        System.out.println("Hello " + paramName +
            ", My name is " + name);
    }
}
```

Analisis kegunaan kode program pada line 6 – 8.

Pada line 6-8, kode program mendefinisikan metode sayHello, yang menerima parameter paramName. Metode ini mencetak pesan sapaan yang menggabungkan nilai dari paramName dan atribut name dari objek Person. Kegunaan dari kode ini adalah untuk memberikan cara bagi objek Person untuk

berinteraksi dengan pengguna atau objek lain dengan cara menyapa, yang dapat berguna dalam aplikasi yang memerlukan interaksi antar pengguna.

```
public class App {
    Run | Debug
    public static void main(String[] args) {
        var person1 = new Person();
        person1.name = "Abdullah";
        person1.address = "Sitoluama";
        // person1.countrt = "Singapura";
        System.out.println(person1.name);
        System.out.println(person1.address);
        System.out.println(person1.country);
        person1.sayHello(paramName:"Kevin");
        Person person2;
        person2 = new Person();
        person2.name = "Kevin";
        person2.sayHello(paramName:"Abdullah");
    }
}
```

Analisisi kegunaan kode program pada line 13 dan 19.

- Pada line 13, kode `System.out.println(person1.name);` digunakan untuk menampilkan nama objek `person1`, yang diatur sebelumnya sebagai "Abdullah". Ini memberikan informasi tentang identitas objek.
- Pada line 19, `person2 = new Person();` mendeklarasikan dan menginisialisasi objek baru `person2` dari kelas `Person`, memungkinkan untuk menyimpan data dan memanggil metode pada objek tersebut. Ini penting untuk membuat dan mengelola lebih dari satu instance dari kelas yang sama.

e. Constructor

```
class Person {
    String name;
    String address;
    final String country = "Indonesia";
    Person(String paramName, String
    paramAddress) {
        name = paramName;
        address = paramAddress;
    }
    void sayHello(String paramName) {
        System.out.println("Hello " + paramName +
        ", My name is " + name);
    }
}
```

Analisis kegunaan kode program pada line 6 – 9.

Pada line 6-9, kode mendefinisikan kelas `Person` yang memiliki atribut `name`, `address`, dan `country`, serta konstruktor yang menerima parameter `paramName` dan `paramAddress`. Konstruktor ini menginisialisasi atribut `name` dan `address` dengan nilai yang diberikan saat objek dibuat, sementara `country` diatur sebagai "Indonesia" dan tidak dapat diubah. Metode `sayHello` memungkinkan objek untuk menyapa orang lain dengan nama yang diberikan, menunjukkan

interaksi antar objek.

```
public class App {
    Run | Debug
    public static void main(String[] args) {
        var person1 = new Person
            (paramName:"Abdullah",
            paramAddress:"Sitoluama");
        System.out.println(person1.name);
        System.out.println(person1.address);
        System.out.println(person1.country);
        person1.sayHello(paramName:"Kevin");
        Person person2;
        // person2 = new Person("Kevin");
        person2 = new Person(paramName:"Kevin",
            paramAddress:"Sitoluama");
        person2.sayHello(paramName:"Abdullah");
    }
}
```

Analisis kegunaan kode program pada line 4 dan 14. Apa yang terjadi jika komentar pada line 13 dihilangkan dan mengapa hasilnya bisa seperti itu.

- Pada line 4, kode var person1 = new Person("Abdullah", "Sitoluama"); digunakan untuk membuat objek person1 dari kelas Person, menginisialisasi atribut name dan address dengan nilai yang diberikan.
- Pada line 14, person2 = new Person("Kevin", "Sitoluama"); juga membuat objek person2, yang menginisialisasi atribut dengan nama "Kevin" dan alamat "Sitoluama".
- Jika komentar pada line 13 dihilangkan dan menjadi person2 = new Person("Kevin");, maka akan terjadi kesalahan kompilasi. Ini karena konstruktor Person memerlukan dua parameter (paramName dan paramAddress), tetapi hanya satu yang diberikan. Tanpa kedua parameter yang diperlukan, program tidak dapat dioperasikan, sehingga menghasilkan error.

f. Constructor Overloading

```
class Person {
    String name;
    String address;
    final String country = "Indonesia";
    Person() {
        this(paramName:null);
    }
    Person(String paramName) {
        this(paramName, paramAddress:null);
    }
    Person(String paramName, String
        paramAddress) {
        name = paramName;
        address = paramAddress;
    }
    void sayHello(String paramName) {
        System.out.println("Hello " + paramName +
            ", My name is " + name);
    }
}
```

Analisis kegunaan kode program pada line 6-8, 10-12, dan 14- 17.

- Kode ini mendefinisikan konstruktor default Person() yang tidak menerima parameter. Dalam konstruktor ini, this(null) dipanggil untuk memanggil konstruktor lain yang menerima satu parameter (paramName). Ini menginisialisasi objek dengan name yang diatur menjadi null dan address akan diatur ke null juga melalui konstruktor berikutnya.
- Kode ini mendefinisikan konstruktor yang menerima satu parameter (paramName). Dengan memanggil this(paramName, null), konstruktor ini menginisialisasi objek dengan name diatur ke nilai parameter dan address diatur ke null. Ini memungkinkan pembuatan objek Person dengan hanya nama, sementara alamat tidak diatur.
- Kode ini mendefinisikan konstruktor yang menerima dua parameter (paramName dan paramAddress). Dalam konstruktor ini, atribut name dan address diinisialisasi dengan nilai parameter yang diberikan. Ini adalah konstruktor utama yang memungkinkan untuk mengatur kedua atribut saat membuat objek Person.

```
public class App {
    Run | Debug
    public static void main(String[] args) {
        var person1 = new Person(paramName:"Abdullah"
        System.out.println(person1.name);
        System.out.println(person1.address);
        System.out.println(person1.country);
        person1.sayHello(paramName:"Kevin");
        Person person2 = new Person(paramName:"Kevin"
        person2.sayHello(paramName:"Abdullah");
        Person person3 = new Person();
    }
}
```

Analisis kegunaan kode program pada line 4, 10, dan 16.

- **Line 4 (var person1 = new Person("Abdullah", "Sitoluama");)**: Ini membuat objek person1 dari kelas Person, dengan dua parameter yaitu nama dan alamat. Kode ini menginisialisasi objek dengan informasi yang spesifik.
- **Line 10 (Person person2 = new Person("Kevin");)**: Di sini, objek person2 diciptakan dengan satu parameter (nama) dan biasanya ini menunjukkan bahwa ada konstruktor lain dalam kelas Person yang hanya memerlukan nama.
- **Line 16 (Person person3 = new Person();)**: Kode ini menciptakan objek person3 tanpa parameter, yang berarti ada konstruktor default dalam kelas Person yang menginisialisasi objek tanpa data awal.

g. Variabel Shadowing

```
class Person {
    String name;
    String address;
    final String country = "Indonesia";
    Person() {
        this(paramName:null);
    }
    Person(String paramName) {
        this(paramName, address:null);
    }
    Person(String name, String address) {
        name = name;
        address = address;
    }
    void sayHello(String name) {
        System.out.println("Hello " + name + ", My
    }
}
```

Analisis kegunaan kode program pada line 14-17 dan 19-21.

- **Line 14-17 (Person() { this(null); }):** Ini adalah konstruktor default yang memanggil konstruktor lain dengan satu parameter (paramName) yang di-set ke null. Hal ini memungkinkan objek Person untuk dibuat tanpa memberikan data awal.
- **Line 19-21 (void sayHello(String name) { System.out.println("Hello " + name + ", My name is " + name); }):** Metode ini mencetak pesan sapaan yang mencakup nama yang diterima sebagai parameter. Namun, ada kesalahan dalam penggunaan parameter name di dalam metode, karena ini bertabrakan dengan atribut name pada kelas.

h. This Keyword

```
class Person {
    String name;
    String address;
    final String country = "Indonesia";
    Person() {
        this(paramName:null);
    }
    Person(String paramName) {
        this(paramName, address:null);
    }
    Person(String name, String address) {
        this.name = name;
        this.address = address;
    }
    void sayHello(String name) {
        System.out.println("Hello " + name + ", My
    }
}
```

Analisis kegunaan kode program pada line 14-17 dan 19-21.

- **Line 14-17 (Person() { this(null); }):** Ini adalah konstruktor default untuk kelas Person. Saat objek Person dibuat tanpa parameter, konstruktor ini

memanggil konstruktor lain yang menerima satu parameter (yaitu paramName) dan menginisialisasi parameter tersebut dengan null. Ini memungkinkan pembuatan objek Person tanpa memberikan informasi nama atau alamat.

- Line 19-21 `(void sayHello(String name) { System.out.println("Hello " + name + ", My name is " + this.name); })`: Metode sayHello menerima parameter name dan mencetak pesan sapaan. Dalam metode ini, this.name merujuk pada atribut name dari objek saat ini, sedangkan name merujuk pada parameter yang diterima oleh metode. Ini memastikan bahwa pesan yang dicetak menyebutkan nama yang diberikan saat memanggil metode, serta nama yang disimpan dalam objek.

i. Inheritance

```
public class App {
    Run | Debug
    public static void main(String[] args) {
        var manager = new Manager();
        manager.name = "Abdullah";
        manager.sayHello(name:"Kevin");
        var vp = new VicePresident();
        vp.name = "Kevin";
        vp.sayHello(name:"Abdullah");
    }
}

public class Manager {
    String name;
    void sayHello(String name) {
        System.out.println("Hi " + name + ", My Name
    }
}
```

Analisis kegunaan kode program pada line 4 dan 8. Amati hasilnya pada terminal.

Baris 4:

- **var:** Kata kunci var digunakan untuk mendeklarasikan variabel dengan tipe inferensi otomatis oleh compiler. Dalam kasus ini, tipe dari variabel manager secara otomatis ditentukan sebagai Manager karena nilai yang diinisialisasi adalah new Manager().
- **new Manager():** Ini adalah instansiasi dari kelas Manager. Dengan kata lain, objek Manager baru dibuat dan disimpan dalam variabel manager.
- **Fungsi baris ini:** Membuat objek baru dari kelas Manager yang nantinya dapat digunakan untuk memanggil metode atau mengakses properti dari objek tersebut.

Baris 8 :

- **var:** Sama seperti di baris 4, var digunakan untuk mendeklarasikan variabel dengan tipe yang diinferensikan secara otomatis oleh compiler. Karena nilai

yang diberikan adalah new VicePresident(), maka tipe dari vp adalah VicePresident.

- **new VicePresident():** Ini adalah instansiasi dari kelas VicePresident. Kelas ini adalah subclass dari Manager, sehingga vp (variabel ini) akan mewarisi atribut dan metode dari kelas Manager, sambil memiliki perilaku tambahan yang didefinisikan khusus untuk kelas VicePresident (jika ada).
- **Fungsi baris ini:** Membuat objek baru dari kelas VicePresident, yang merupakan turunan dari kelas Manager, dan menyimpan objek tersebut dalam variabel vp. Objek ini kemudian dapat digunakan untuk memanggil metode dan mengakses properti yang diwarisi dari Manager maupun metode/properti yang didefinisikan di kelas VicePresident

```
public class VicePresident extends Manager {  
    }  
}
```

Analisis kegunaan kode program pada line 1.

- **public:** Modifier ini menunjukkan bahwa kelas VicePresident dapat diakses oleh semua kelas lain. Ini berarti kelas ini bisa digunakan di mana saja dalam program, tidak terbatas pada paket yang sama.
- **class:** Kata kunci class digunakan untuk mendeklarasikan sebuah kelas baru. Dalam hal ini, kelas yang dibuat adalah VicePresident.
- **VicePresident:** Ini adalah nama kelas yang sedang dibuat. Kelas ini akan mewakili entitas tertentu yang bisa berupa objek dengan atribut dan metode.
- **extends Manager:** Kata kunci extends menunjukkan bahwa kelas VicePresident adalah subclass dari kelas Manager. Artinya, VicePresident akan mewarisi semua atribut dan metode dari kelas Manager. Dengan kata lain, VicePresident adalah turunan atau spesialisasi dari Manager, dan mungkin akan menambahkan beberapa fungsi tambahan di atasnya.

j. Method Overriding

```
java > ...  
public class Manager {  
    String name;  
    void sayHello(String name) {  
        System.out.println("Hi " + name + ", My  
        Name is " + this.name + "  
        (Manager)");  
    }  
}  
  
public class VicePresident extends Manager {  
    void sayHello(String name) {  
        System.out.println("Hi " + name + ", My  
        Name is " + this.name + "  
        (Vice President)");  
    }  
}
```

Analisis pada class apa terjadi overriding dan pada line berapa terjadinya overriding.

1. **Class VicePresident:** Kelas ini mewarisi (extends) kelas Manager. Jika di kelas Manager terdapat metode sayHello(String name), maka metode ini akan di-override dalam kelas VicePresident.

2. Metode sayHello(String name):

- Pada baris 2, kita melihat metode sayHello(String name) didefinisikan dalam kelas VicePresident.
- Jika metode yang sama juga ada di kelas Manager, maka metode ini adalah bentuk *overriding*. Dengan kata lain, metode ini diimplementasikan ulang di kelas VicePresident untuk memberikan perilaku yang berbeda (menambahkan "(Vice President)" di akhir kalimat).
- Overriding terjadi karena subclass VicePresident menggunakan metode sayHello dengan nama dan parameter yang sama seperti di superclass Manager, namun mengubah implementasinya.

k. Super Keyword

```
public class Shape {
    int getCorner() {
        return 0;
    }
    class Rectangle extends Shape {
        int getCorner() {
            return 4;
        }
        int getParentCorner() {
            return super.getCorner();
        }
    }
}
```

Analisis kegunaan kode program pada line 16.

- **Keyword super:** super digunakan untuk memanggil metode atau properti dari superclass (kelas induk) dari kelas yang sedang berjalan. Dalam kasus ini, super.getCorner() memanggil metode getCorner() yang didefinisikan di kelas Shape.
- **Fungsi dari getParentCorner():** Metode getParentCorner() dalam kelas Rectangle bertujuan untuk mengakses versi metode getCorner() yang berada di superclass (Shape). Ini memungkinkan kelas Rectangle untuk memanfaatkan logika atau nilai dari metode getCorner() milik kelas Shape, meskipun telah di-override di kelas Rectangle.
- **Kegunaan pada baris 16:**
 - Baris ini mengembalikan nilai 0, karena metode getCorner() pada kelas Shape mengembalikan nilai 0.
 - Jadi, meskipun Rectangle memiliki metode getCorner() yang mengembalikan nilai 4, dengan menggunakan super.getCorner(), metode getParentCorner() akan mengembalikan nilai 0, yang berasal dari implementasi metode getCorner() di kelas Shape.

```

public class App {
    Run | Debug
    public static void main
        (String[] args) {
        var shape = new Shape();
    System.out.println(shape.
        getCorner());
    var rectangle = new Rectangle
        ();
    System.out.println(rectangle.
        getCorner());
    System.out.println(rectangle.
        getParentCorner());
    }
}

```

Analisis kegunaan kode program pada line 8 dan 9. :

- **Baris 8: System.out.println(rectangle.getCorner());**
 - Baris ini memanggil metode getCorner() dari objek rectangle yang merupakan instansiasi dari kelas Rectangle.
 - Metode getCorner() telah di-override dalam kelas Rectangle, sehingga yang akan dijalankan adalah versi metode yang ada di kelas Rectangle, yang mengembalikan nilai **4**.
 - Keluaran pada baris ini adalah **4**, karena kelas Rectangle mengembalikan nilai tersebut pada metode getCorner().
- **Baris 9: System.out.println(rectangle.getParentCorner());**
 - Baris ini memanggil metode getParentCorner() dari objek rectangle.
 - Metode getParentCorner() menggunakan super.getCorner() untuk memanggil versi metode getCorner() yang ada di kelas induk (Shape), yang mengembalikan nilai **0**.
 - Keluaran pada baris ini adalah **0**, karena super.getCorner() merujuk pada metode getCorner() di kelas Shape

1. Super Constructor

```

public class Manager {
    String name;
    String company;
    Manager(String name) {
        this(name, company:null);
    }
    Manager(String name, String company) {
        this.name = name;
        this.company = company;
    }
    void sayHello(String name) {
        System.out.println("Hi " + name + ", My
        Name is " + this.name + "
        (Manager)");
    }
}

```

```
public class VicePresident {
    VicePresident(String name) {
        super(name);
    }
    void sayHello(String name) {
        System.out.println("Hi " + name + ", My
Name is " + this.name + "
(Vice President)");
    }
}
```

Analisis kegunaan kode program pada line 4.

- **Memanggil Konstruktor Kelas Induk (Superclass)**: Line super(name); dipakai untuk memanggil konstruktor dari kelas induk (parent class) dari VicePresident. Hal ini berarti VicePresident adalah subclass dari kelas yang lebih umum (mungkin President atau kelas lain) yang memiliki konstruktor dengan parameter name.
- **Pewarisan Sifat (Inheritance)**: Karena kelas VicePresident mewarisi dari kelas induk, penggunaan super(name) ini memungkinkan objek VicePresident untuk mengakses atau menginisialisasi properti yang didefinisikan di kelas induknya, misalnya properti name yang ada di superclass.
- **Inisialisasi Properti Kelas Induk**: Line ini penting karena memastikan bahwa atribut yang diturunkan dari kelas induk diinisialisasi dengan benar. Jika konstruktor kelas induk mengharapkan input berupa name, maka super(name) meneruskannya.

```
public class App {
    Run | Debug
    public static void main(String[] args) {
        var manager = new Manager(name:"Abdullah",
        company:"ITDel");
        manager.sayHello(name:"Kevin");
        var vp = new VicePresident(name:"Kevin");
        vp.sayHello(name:"Abdullah");
    }
}
```

Analisis kegunaan kode program pada line 4 dan 7.

Line 4 :

- **Deklarasi dan Inisialisasi Objek**: Line ini mendeklarasikan sebuah variabel manager dan menginisialisasi objek baru dari kelas Manager dengan menggunakan kata kunci var (type inference). Java akan secara otomatis menentukan tipe data dari variabel ini berdasarkan objek yang diciptakan, yaitu Manager.
- **Membuat Objek**: Objek Manager dibuat dengan memanggil konstruktor dari kelas Manager yang menerima dua parameter, yaitu name (dalam hal ini "Abdullah") dan departemen atau informasi lainnya (dalam hal ini

"ITDel"). Objek ini sekarang memiliki semua atribut dan metode dari kelas Manager.

- **Penggunaan Nama dan Departemen:** Dengan menggunakan nama "Abdullah" dan mungkin departemen "ITDel", objek ini siap digunakan untuk menjalankan metode yang dimiliki oleh kelas Manager, termasuk memanggil metode sayHello

Line 7 :

- **Deklarasi dan Inisialisasi Objek:** Sama seperti di line 4, line ini mendeklarasikan variabel vp yang secara otomatis akan bertipe VicePresident berdasarkan objek yang diinisialisasi. Objek tersebut adalah instance dari kelas VicePresident dengan nama "Kevin".
- **Membuat Objek VicePresident:** Objek VicePresident dibuat dengan memanggil konstruktor dari kelas VicePresident, yang hanya menerima satu parameter, yaitu name. Pada line ini, nama yang diberikan adalah "Kevin".
- **Siap untuk Menjalankan Metode:** Objek vp yang sudah diinisialisasi sekarang dapat menggunakan semua metode yang ada di kelas VicePresident, termasuk memanggil metode sayHello.

m. Object Class

```
public class App {
    Run|Debug
    public static void main(String[] args) {
        var manager = new Manager(name:"Abdullah",
        company:"ITDel");
        manager.sayHello(name:"Kevin");
        var vp = new VicePresident(name:"Kevin");
        vp.sayHello(name:"Abdullah");
        System.out.println(manager);
        System.out.println(manager.toString());
        System.out.println(vp);
        System.out.println(vp.toString());
    }
}
```

Analisis apa tujuan dari kode program pada line 10, 11, 12, dan 13.

Line 10 :

- **Tujuan:** Pada baris ini, ketika objek manager dilewatkan langsung ke System.out.println(), Java secara otomatis memanggil method toString() dari objek tersebut. Jika kelas Manager telah mengoverride method toString() (yang diwarisi dari Object), maka output yang ditampilkan adalah representasi string dari objek manager sesuai dengan implementasi di dalam kelas Manager.
- **Default Behavior:** Jika method toString() belum dioverride dalam kelas Manager, maka yang ditampilkan adalah string default dari Object, yang biasanya berupa nama kelas dan nilai hash code objek (misalnya, Manager@6d06d69c).

Line 11:

- **Tujuan:** Line ini secara eksplisit memanggil method `toString()` pada objek manager. Fungsinya sama dengan line 10, tetapi di sini Anda memanggil `toString()` secara manual. Jika `toString()` telah di-override dalam kelas Manager, output akan berupa representasi string dari objek manager.
- **Perbedaan dengan Line 10:** Meski secara fungsional sama, line ini lebih eksplisit dibandingkan dengan line 10. Pemanggilan `toString()` secara langsung memberikan kejelasan bahwa Anda memang bermaksud mendapatkan representasi string dari objek.

Line 12:

- **Tujuan:** Sama seperti pada line 10, tetapi kali ini objek yang diproses adalah vp, yang merupakan instance dari kelas VicePresident. Jika VicePresident telah mengoverride method `toString()`, maka representasi string yang ditampilkan akan sesuai dengan implementasi yang dibuat di kelas tersebut.
- **Default Behavior:** Jika `toString()` tidak di-override dalam kelas VicePresident, Java akan menggunakan implementasi `toString()` dari kelas induk atau implementasi default dari Object.

Line 13:

- **Tujuan:** Sama seperti pada line 11, tetapi untuk objek vp. Line ini secara eksplisit memanggil method `toString()` dari objek VicePresident. Jika method ini di-override dalam kelas VicePresident, representasi string yang ditampilkan akan berupa hasil dari implementasi `toString()` yang dibuat di kelas tersebut.

n. Polymorphism

```
public class Employee {
    String name;
    Employee(String name) {
        this.name = name;
    }
    void sayHello(String name) {
        System.out.println("Hi " + name + ", My
Name is " + this.name + "
(Employee)");
    }
}
```

```

public class Manager extends Employee {
    String name;
    String company;
    Manager(String name) {
        this(name, company:null);
    }
    Manager(String name, String company) {
        super(name);
    }
    this.name = name;
    this.company = company;
}
void sayHello(String name) {
    System.out.println("Hi " + name + ", My
Name is " + this.name + "
(Manager)");
}
}
public class App {
    Run|Debug
    public static void main(String[] args) {
        Employee employee = new Employee
        (name:"Abdullah");
        employee.sayHello(name:"Kevin");
        employee = new Manager(name:"Abdullah");
        employee.sayHello(name:"Kevin");
        employee = new VicePresident
        (name:"Abdullah");
        employee.sayHello(name:"Kevin");
        sayHello(new Employee(name:"Abdullah"));
    }
    sayHello(new Manager(name:"Kevin"));
    sayHello(new VicePresident
    (name:"Aprialdy"));
}
static void sayHello(Employee employee) {
    System.out.println("Hello " + employee.
    name);
}
}

```

Analisis tujuan dari kode program pada line 4, 7, 10, 13, 14, 15, dan 18 – 20.

- Line 4: Membuat objek employee dari kelas Employee dengan nama "Abdullah". Pada baris ini, objek employee dibuat sebagai instance dari kelas Employee.
- Line 7 : Mengubah referensi employee menjadi objek dari kelas Manager dengan nama "Abdullah". Ini adalah contoh **polimorfisme**, di mana objek employee yang awalnya merupakan instance dari kelas Employee, sekarang menjadi instance dari subclass Manager.
- Line 10 : Mengubah referensi employee menjadi objek dari kelas Manager dengan nama "Abdullah". Ini adalah contoh **polimorfisme**, di mana objek employee yang awalnya merupakan instance dari kelas Employee, sekarang menjadi instance dari subclass Manager.
- Line 13 : Memanggil metode sayHello dengan parameter objek baru dari kelas Employee bernama "Abdullah". Metode ini akan mencetak "Hello Abdullah" ke console.
- Line 14 : Memanggil metode sayHello dengan parameter

objek baru dari kelas Manager bernama "Kevin".

Karena Manager adalah subclass dari Employee, metode sayHello akan mencetak "Hello Kevin".

- Line 15 : Memanggil metode sayHello dengan parameter objek baru dari kelas VicePresident bernama "Aprialdy". Sama seperti sebelumnya, akan dicetak "Hello Aprialdy" karena VicePresident adalah subclass dari Employee.
- Line 18-20 : Mendefinisikan metode sayHello yang menerima parameter berupa objek dari kelas Employee atau subclass-nya. Metode ini mencetak pesan "Hello" diikuti dengan atribut name dari objek Employee yang diberikan.

o. Type Check dan Casts

```
public class App {
    public static void main(String[] args) {
        employee.sayHello(name:"Kevin");
        employee = new VicePresident(name:"Abdullah");
        employee.sayHello(name:"Kevin");
        sayHello(new Employee(name:"Abdullah"));
        sayHello(new Manager(name:"Kevin"));
        sayHello(new VicePresident(name:"Aprialdy"));
    }
    static void sayHello(Employee employee) {
        if (employee instanceof VicePresident) {
            VicePresident vicePresident = (VicePresident) employee;
            System.out.println("Hello VP " + vicePresident.name);
        } else if (employee instanceof Manager) {
            Manager manager = (Manager) employee;
            System.out.println("Hello Manager " + manager.name);
        } else {
            System.out.println("Hello " + employee.name);
        }
    }
}
```

Analisis tujuan dari kode program pada line 19, 20, 22 dan 23.

- Line 19 : Mengecek apakah objek employee yang diteruskan ke metode sayHello merupakan instance dari kelas VicePresident. Ini menggunakan operator instanceof untuk memastikan apakah objek tersebut berasal dari kelas VicePresident atau subclass-nya. Jika kondisi ini terpenuhi, maka program akan menjalankan blok kode yang ada di dalamnya, yang melakukan **downcasting** ke tipe VicePresident.
- Line 20 : Melakukan **downcasting** dari objek employee yang bertipe Employee ke tipe VicePresident. Ini memungkinkan akses langsung ke atribut atau metode khusus dari kelas VicePresident. Setelah downcast, objek vicePresident adalah referensi ke objek yang sebenarnya dari tipe VicePresident.
- Line 22 : Mengecek apakah objek employee merupakan instance dari kelas Manager. Jika objek employee adalah dari tipe Manager, maka blok kode berikutnya akan dijalankan. Seperti pada line 19, ini juga menggunakan **polimorfisme** untuk menentukan tipe spesifik dari objek pada runtime.
- Line 23 : Melakukan **downcasting** dari objek employee ke tipe Manager. Sama seperti pada line 20, ini

memungkinkan akses ke atribut atau metode yang mungkin hanya ada di kelas Manager. Objek manager sekarang adalah referensi ke objek Manager.

p. Variable Hiding

```
public class Parent {
    String name;
    void info() {
        System.out.println("Ini adalah class Parent");
    }
}
class Child extends Parent {
    String name;
    void info() {
        System.out.println("Ini adalah class Child");
        System.out.println("Nama parent class adalah " + super.
    name);
}
}
```

Analisis apakah field pada line 2 dan 10 sama. Apa tujuan kode program pada line 14.

- Line 2 : Field ini dideklarasikan dalam kelas **Parent**. Artinya, setiap objek yang merupakan instance dari kelas Parent (atau subclass-nya) akan memiliki field **name** yang terkait dengan objek Parent.
- Line 10 : Field ini dideklarasikan dalam kelas **Child**, yang merupakan subclass dari Parent. Field ini **tidak sama** dengan field **name** yang ada di kelas Parent, meskipun namanya sama. Karena field ini dideklarasikan ulang di kelas Child, ia akan **menyembunyikan** (hide) field **name** yang ada di kelas Parent dalam konteks objek Child.
- Line 14 : Menggunakan kata kunci **super** untuk mengakses field **name** dari **kelas Parent**. Karena kelas Child mendeklarasikan ulang field **name**, jika hanya menggunakan **name** saja, program akan mengakses field dari kelas Child. Oleh karena itu, dengan menggunakan **super.name**, program secara eksplisit mengakses field **name** dari kelas Parent.

```
public class App {
    Run | Debug
    public static void main(String[] args) {
        Child child = new Child();
        child.name = "Abdullah";
        child.info();
        System.out.println(child.name);
        Parent parent = child; // Boleh langsung tanpa casts,
        karena class
        child anak dari class parent
        parent.info();
        System.out.println(parent.name);
    }
}
```

Analisis hasil kode program pada line 7 dan 11 dan mengapa hasilnya bisa seperti itu.

- Line 7 : Pada line ini, objek **child** dari kelas **Child** memiliki field **name** yang sudah **di-override** (dideklarasikan ulang) di dalam kelas **Child**. Jadi,

ketika Anda mengakses child.name, program akan mengakses **field name milik Child**. Karena pada baris sebelumnya (child.name = "Abdullah";), Anda menetapkan nilai name menjadi "Abdullah", maka yang dicetak pada line 7 adalah "Abdullah".

- Line 11 : Pada line ini, meskipun objek parent sebenarnya mengacu pada objek child (karena parent = child), akses terhadap field name akan dilakukan berdasarkan **tipe referensi**. Dalam hal ini, referensi parent bertipe Parent, jadi yang diakses adalah **field name dari kelas Parent**.

Field name dalam kelas Parent **tidak diinisialisasi** secara eksplisit dalam kode ini, sehingga nilainya secara default adalah **null** (karena name bertipe String, dan nilai default untuk variabel objek yang belum diinisialisasi adalah null).

Meskipun referensi parent mengacu pada objek Child, aturan dalam Java mengatakan bahwa akses terhadap field didasarkan pada tipe deklarasi, bukan pada tipe objek sebenarnya. Oleh karena itu, meskipun parent mengacu pada objek Child, akses parent.name mengacu pada field name di kelas Parent, yang belum diatur, sehingga hasilnya adalah null.

Alasan hasilnya :

- Akses ke **field** di Java ditentukan oleh **tipe referensi**, bukan oleh tipe objek yang sebenarnya (dalam hal ini, Parent dan Child). Ini disebut **field hiding** (penyembunyian field).
- Akses ke **metode** (info()) mengikuti **polimorfisme** di mana metode dipanggil berdasarkan tipe objek sebenarnya (dalam kasus ini, Child), sedangkan akses ke **field** didasarkan pada tipe referensi (dalam kasus ini, Parent).

q. Package

```
package delcom.model;
public class Product {
    String name;
    int price;
    public Product(String name, int price) {
        this.name = name;
        this.price = price;
    }
}
```

Analisis tujuan kode program pada line 1.

Deklarasi Package: Baris ini menyatakan bahwa kelas Product berada di dalam **package** bernama delcom.model. Package dalam Java digunakan untuk mengelompokkan kelas-kelas yang memiliki hubungan atau fungsionalitas yang mirip ke dalam satu unit, sehingga memudahkan pengorganisasian kode program.

r. Access Modifier

```
package delcom.model;
public class Product {
    private String name;
    protected int price;
    public Product(String name, int price) {
        this.name = name;
        this.price = price;
    }
}
```

Analisis tujuan kode program pada line 4 dan 5. Apakah field pada line 4 dapat diakses di class TestProduct.java

- Line 4 : Field name dideklarasikan dengan akses modifier **private**. Artinya, field ini hanya dapat diakses dari **dalam kelas Product** itu sendiri. Field ini tidak dapat diakses secara langsung dari luar kelas, bahkan oleh kelas lain yang berada dalam package yang sama, seperti TestProduct.
- Line 5 : Field price dideklarasikan dengan akses modifier **protected**. Ini berarti field price dapat diakses oleh:
 - Kelas Product itu sendiri.
 - Kelas lain dalam package yang sama (dalam hal ini, kelas TestProduct karena berada di package delcom.model).
 - Kelas yang merupakan **subclass** (pewaris) dari Product, baik di package yang sama maupun di package yang berbeda.
- Akses : **Tidak**, field name yang dideklarasikan sebagai private di kelas Product **tidak dapat diakses** secara langsung di kelas TestProduct, meskipun kedua kelas berada di package yang sama. Akses langsung hanya diperbolehkan dari dalam kelas Product itu sendiri

```
package delcom.model;
public class TestProduct {
    Run | Debug
    public static void main(String[] args) {
        Product product = new Product(name:"Sari Gandum",
            price:10_000);
        product.price = 12_000;
        System.out.println(product.price);
    }
}
```

Analisis kenapa field price dapat diakses pada line 6.

- **Modifier protected:**
 - Dalam Java, field yang dideklarasikan dengan modifier protected dapat diakses oleh:
 - **Kelas yang sama**.
 - **Kelas dalam package yang sama**.
 - **Subclass dari kelas Product** (di mana pun lokasinya, baik di package yang sama atau berbeda).
- **Kelas TestProduct dan Product Berada dalam Package yang Sama:**
 - Kelas Product dan TestProduct keduanya berada dalam package **delcom.model**, sehingga TestProduct memiliki akses langsung ke field price yang memiliki modifier protected.

Karena TestProduct dan Product berada dalam package yang sama, akses terhadap field price diperbolehkan tanpa perlu melakukan inheritance (pewarisan) atau penggunaan getter/setter.

- **Modifier protected Membolehkan Akses Package-Level:**
 - **Package-level access** adalah salah satu fitur dari protected, yang mengizinkan akses ke anggota yang dideklarasikan dengan protected di kelas mana pun yang ada dalam package yang sama. Pada line 6, product.price = 12_000; berhasil dijalankan karena TestProduct adalah bagian dari package yang sama dengan Product.

s. Import

```
package controller;

public class A1Controller {

}

package controller;

public class A2Controller {

}

package delcom.app;
import delcom.controller.*;
import delcom.model.Product;
public class App {
    A1Controller a1 = new A1Controller();
    A2Controller a2 = new A2Controller();
    Product product = new Product("Indomie", 3_000);
    // product.price = 5_000;
}
```

Analisis tujuan kode program pada line 3 dan 4. Apa yang terjadi jika komentar pada line 12 di hilangkan dan mengapa hal tersebut dapat terjadi.

- Line 3 : Pada baris ini, sebuah objek dari kelas **A1Controller** diciptakan dan disimpan dalam variabel a1. Ini adalah proses **instansiasi** dari kelas A1Controller. Tujuannya adalah untuk membuat objek yang dapat digunakan untuk memanggil metode atau mengakses properti yang ada di dalam kelas A1Controller.
- Line 4 : Sama seperti pada line 3, pada baris ini objek dari kelas **A2Controller** dibuat dan disimpan dalam variabel a2. Ini memungkinkan program untuk menggunakan objek a2 untuk memanggil metode atau mengakses properti di dalam kelas A2Controller.
- Line 12 :
 - **Field price pada Kelas Product:**

- Di dalam kelas Product, field price dideklarasikan dengan akses **protected**
- **Field protected** hanya dapat diakses:
 - Dari dalam kelas itu sendiri.
 - Oleh kelas-kelas **turunan** (subclass) dari kelas tersebut, di mana pun mereka berada.
 - Oleh **kelas-kelas lain di dalam package yang sama**.
- **Akses dari Kelas App:**
 - Kelas App berada di dalam **package delcom.app**, sementara kelas Product berada di dalam **package delcom.model**.
 - Karena **protected** tidak mengizinkan akses lintas-package, kecuali melalui subclass, maka kelas App tidak dapat mengakses secara langsung field price di kelas Product.
- **Error yang Akan Terjadi:** Jika komentar di line 12 dihapus, program akan menghasilkan **error kompilasi**. Ini karena **protected** field price tidak dapat diakses langsung dari luar package delcom.model oleh kelas yang tidak merupakan subclass dari Product.

t. Abstract Class

```
public abstract class Location {
    public String name;
}
class City extends Location {
```

Analisis tujuan kode program pada line 1.

- **Deklarasi Kelas Abstrak:**
 - Line 1 mendeklarasikan kelas **Location** sebagai **kelas abstrak** dengan menggunakan kata kunci **abstract**.
 - **Kelas abstrak** dalam Java adalah kelas yang tidak dapat diinstansiasi secara langsung. Tujuannya adalah untuk menyediakan **kerangka dasar** atau **blueprint** bagi kelas turunannya, di mana kelas turunan ini harus mengimplementasikan metode-metode abstrak (jika ada) yang dideklarasikan dalam kelas abstrak.
- **Tujuan Kelas Location:**
 - Kelas Location mungkin digunakan sebagai **kelas dasar** yang menyediakan struktur untuk kelas-kelas turunan seperti City. Kelas ini bisa berisi properti yang umum bagi semua tipe lokasi (seperti name), tetapi tidak secara spesifik mendefinisikan implementasi detail untuk objek Location itu sendiri.
 - Karena kelas ini abstrak, berarti **tujuan utamanya** adalah untuk digunakan sebagai **superclass** bagi kelas lain yang lebih spesifik, seperti City.
- **Mengapa Kelas Ini Abstrak:**
 - Kelas dibuat abstrak ketika **tidak masuk akal untuk membuat objek langsung dari kelas tersebut**. Sebagai contoh, mungkin tidak ada objek langsung yang hanya disebut "Location", tetapi ada objek turunan seperti

City, Country, dll., yang akan mengisi detail spesifik.

- Kelas abstrak juga bisa berisi **metode abstrak** yang harus diimplementasikan oleh kelas turunan. Namun, dalam contoh ini, kelas Location hanya berisi field name, jadi tidak ada metode abstrak.

```
public class App {
    Run | Debug
    public static void main(String[] args) {
        // var location = new Location();
        var city = new City();
        city.name = "Balige";
        System.out.println(city.name);
    }
}
```

Analisis apa yang terjadi jika kode program pada line 4 dihilangkan komentarnya.

Jika komentar pada **line 4** dihilangkan, program akan menghasilkan **error kompilasi** karena kelas Location adalah kelas abstrak dan tidak dapat diinstansiasi secara langsung. Anda hanya bisa menginstansiasi kelas konkret seperti City, yang merupakan subclass dari Location.

u. Abstract Method

```
public abstract class Animal {
    public String name;
    public abstract void run();
}
```

Analisis kegunaan kode program pada line 1 dan 4.

- **Line 1:** Mendeklarasikan **Animal** sebagai kelas abstrak yang tidak dapat diinstansiasi dan digunakan sebagai dasar untuk kelas turunan. Ini menyediakan struktur umum untuk semua hewan.
- **Line 4:** Mendeklarasikan **run** sebagai metode abstrak yang harus diimplementasikan oleh semua subclass dari Animal, memastikan bahwa setiap jenis hewan memiliki cara khusus untuk melakukan aksi **run** dan memungkinkan penggunaan polimorfisme dalam kode.

```
public class Cat extends Animal {
    public void run() {
        System.out.println("Cat " + name + " is run");
    }
}
```

Analisis apa yang terjadi apabila kode program pada line 2 sampai 4 dikomentari.

Jika kode pada **line 2 sampai 4** dikomentari, kelas **Cat** tidak

akan dapat dikompilasi karena tidak mengimplementasikan metode abstrak run yang dideklarasikan dalam kelas **Animal**. Ini akan menghasilkan error kompilasi, mengharuskan **Cat** untuk menyediakan implementasi untuk metode run, atau jika tidak, dideklarasikan sebagai kelas abstrak. Penghilangan implementasi ini akan menghilangkan kemampuan spesifik Cat untuk mendefinisikan perilaku run, sehingga menyebabkan kehilangan fungsionalitas penting dari kelas tersebut.

```
public class App {
    Run | Debug
    public static void main(String[] args) {
        Animal animal = new Cat();
        animal.name = "Anggora";
        animal.run();
    }
}
```

Analisis kegunaan kode program pada line 4, 5, dan 6

- **Line 4:** Menunjukkan penggunaan polimorfisme dengan menginstansiasi objek dari subclass (Cat) tetapi menyimpannya dalam variabel bertipe superclass (Animal).
- **Line 5:** Mengatur field name untuk memberikan identitas kepada objek animal, memungkinkan pengenal spesifik pada objek tersebut.
- **Line 6:** Memanggil metode run, yang akan mengeksekusi implementasi spesifik dari kelas Cat, memanfaatkan mekanisme polimorfisme untuk menjalankan metode yang tepat meskipun objek disimpan dalam variabel tipe superclass.

v. Getter dan Setter

```
public class Category {
    private String id;
    private boolean expensive;
    public String getId() {
        return id;
    }
    public void setId(String id) {
        if (id != null) {
            this.id = id;
        }
    }
    public boolean isExpensive() {
        return expensive;
    }
    public void setExpensive(boolean expensive) {
        this.expensive = expensive;
    }
}
```

Analisis kegunaan kode program pada line 5-7, 9-13, 15-17, dan 19-21.

- **Line 5-7:** Mendeklarasikan getter untuk field id, mendukung prinsip enkapsulasi dan menyediakan cara aman untuk mengakses nilai id.

- **Line 9-13:** Mendeklarasikan setter untuk field id yang mencakup validasi input, memastikan bahwa nilai yang tidak valid tidak disimpan.
- **Line 15-17:** Mendeklarasikan getter untuk field expensive, memberikan informasi tentang status mahalnya kategori.
- **Line 19-21:** Mendeklarasikan setter untuk field expensive, yang memungkinkan pengguna untuk mengatur status mahal kategori dengan cara yang terstruktur.

```
public class App {
    Run | Debug
    public static void main(String[] args) {
        var category = new Category();
        category.setId(id:"123456");
        category.setId(id:null);
        System.out.println(category.getId());
    }
}
```

Analisis kegunaan kode program pada line 5, 6, dan 8.

Apakah output dari kode program pada line 8.

- **Line 5:** Menginstansiasi objek category dari kelas Category.
- **Line 6:** Menetapkan nilai "123456" ke field id menggunakan setter.
- **Line 8:** Mengambil dan mencetak nilai id, yang akan menghasilkan output:

123456

w. Interface

```
public interface Car extends HasBrand, IsMaintenance {
    void drive();
    int getTier();
}
```

Analisi kegunaan kode program pada line 1.

- **Line 1** mendeklarasikan antarmuka Car, yang merupakan kontrak bagi semua kendaraan, dengan mewajibkan implementasi metode drive() dan getTier().
- Dengan mewarisi dari antarmuka **HasBrand** dan **IsMaintenance**, antarmuka Car menciptakan struktur yang lebih kaya dan mendalam untuk berbagai jenis kendaraan.
- Desain ini sangat penting dalam pengembangan sistem perangkat lunak yang modular dan berbasis komponen, memungkinkan kelas-kelas berbeda untuk berbagi fungsi tanpa harus terikat pada hierarki kelas yang kaku.

```
public class Avanza implements Car {
    public void drive() {
        System.out.println("Avanza Drive");
    }
    public int getTier() {
        return 4;
    }
}
```

Analisis kegunaan kode program pada line 3-5 dan 7-9.

- **Line 3-5:** Kelas Avanza diimplementasikan untuk menyediakan metode `drive()`, yang menunjukkan perilaku mengemudikan mobil. Ini menciptakan keterhubungan antara kelas Avanza dan antarmuka `Car`, mendukung prinsip pemrograman berorientasi objek.
- **Line 7-9:** Metode `getTier()` memberikan informasi tambahan tentang objek Avanza, yang mengindikasikan tingkat atau kategori tertentu. Ini membantu untuk mengklasifikasikan kendaraan dalam konteks yang lebih besar.

```
public class App {
    Run | Debug
    public static void main(String[] args) {
        Car car = new Avanza();
        System.out.println(car.getTier());
        car.drive();
    }
}
```

x. Interface Inheritance

```
public class App {
    Run | Debug
    public static void main(String[] args) {
        Car car = new Avanza();
        System.out.println(car.getTier());
        System.out.println(car.getBrand());
        // System.out.println(car.getPrice());
        System.out.println(((Avanza) car).getPrice());
        car.drive();
    }
}
```

Analisis tujuan kode program pada line 8. Analisis apa yang terjadi jika kode program pada line 7 dihilangkan komentarnya.

```
public interface HasBrand {
    String getBrand();
}
```

```
public interface IsMaintenance {
    boolean isMaintenance();
}

public interface Car extends HasBrand, IsMaintenance {
    void drive();
    int getTier();
}
```

Analisi kegunaan kode program pada line 1.

- **Line 1** mendeklarasikan antarmuka Car, yang merupakan kontrak untuk semua kendaraan, dengan mengharuskan implementasi metode drive() dan getTier().
- Dengan mewarisi dari antarmuka **HasBrand** dan **IsMaintenance**, antarmuka Car menggabungkan berbagai perilaku, mendorong desain yang bersih dan terorganisir dalam aplikasi.
- Konsep ini sangat penting dalam desain sistem yang modular dan berbasis komponen, memungkinkan kelas-kelas berbeda untuk berbagi fungsi tanpa harus terikat pada hierarki kelas yang kaku.

```
public class Avanza implements Car, Catalog {
    public void drive() {
        System.out.println("Avanza Drive");
    }
    public int getTier() {
        return 4;
    }
    public String getBrand() {
        return "Toyota";
    }
    public boolean isMaintenance() {
        return false;
    }
    public int getPrice() {
        return 100_000_000;
    }
}
```

Analisis kegunaan kode program pada line 3-5 dan 7-9.

- **Line 3-5:** Kelas Avanza diimplementasikan untuk menyediakan metode drive(), yang menunjukkan perilaku mengemudikan mobil. Ini menciptakan keterhubungan antara kelas Avanza dan antarmuka Car, mendukung prinsip pemrograman berorientasi objek.
 - **Line 7-9:** Keduanya mendefinisikan metode getTier() dan getBrand() yang memberikan informasi tambahan tentang objek Avanza, memenuhi kontrak dari antarmuka Catalog.
- y. Default Method

```
public interface Car extends HasBrand, IsMaintenance {
    void drive();
    int getTier();
    default boolean isBig() {
        return false;
    }
    default boolean isMaintenance() {
        return false;
    }
}
```

Analisis kegunaan kode program pada line 6-8 dan 10-12.

- **Line 6-8:** Metode default isBig() memberikan implementasi standar untuk menentukan apakah mobil besar. Ini meningkatkan fleksibilitas dalam pengembangan dengan memungkinkan kelas yang mengimplementasikan antarmuka untuk menggunakan logika standar atau menyesuaikannya sesuai kebutuhan mereka.
- **Line 10-12:** Metode default isMaintenance() menyediakan informasi tentang kebutuhan pemeliharaan kendaraan secara umum. Ini memastikan bahwa semua implementasi dari Car memiliki akses ke informasi ini tanpa perlu mendefinisikannya setiap kali, mendukung prinsip DRY (Don't Repeat Yourself).

```
public class Bus implements Car {
    public void drive() {
        System.out.println("Bus Drive");
    }
    public int getTier() {
        return 8;
    }
    public String getBrand() {
        return "Hino";
    }
    public boolean isMaintenance() {
        return false;
    }
    public boolean isBig() {
        return true;
    }
}
```

Analisis kegunaan kode program pada line 19-21.

- **Line 19-21:** Implementasi metode isBig() dalam kelas Bus menyediakan detail spesifik bahwa bus merupakan kendaraan besar. Ini menambah konteks yang berguna untuk aplikasi yang bekerja dengan berbagai jenis kendaraan dan membantu mematuhi prinsip desain antarmuka yang mendukung keberagaman perilaku di antara implementasi yang berbeda.
- Dengan cara ini, kelas Bus tidak hanya memenuhi kontrak dari antarmuka Car, tetapi juga memberikan informasi tambahan yang mungkin dibutuhkan dalam konteks aplikasi yang lebih luas, menjadikannya lebih bermanfaat dan informatif bagi pengembang lain yang menggunakan atau mengelola kelas ini.

```
public class App {
    Run | Debug
    public static void main(String[] args) {
        Car car = new Avanza();
        System.out.println(car.isBig());
        car.drive();
        car = new Bus();
        System.out.println(car.isBig());
        car.drive();
    }
}
```

Analisis hasil kode program pada line 5 dan 9.

- **Line 5:** Memanggil car.isBig() pada objek Avanza menghasilkan output false karena Avanza menggunakan implementasi default dari antarmuka Car, yang mengembalikan false.
 - **Line 9:** Memanggil car.isBig() pada objek Bus menghasilkan output true karena Bus telah mengoverride metode tersebut untuk mengembalikan nilai true, menunjukkan bahwa bus dianggap sebagai kendaraan besar.
- z. `toString()` Method

```
class Person {
    String name;
    String address;
    final String country = "Indonesia";
    Person() {
        this(paramName:null);
    }
    Person(String paramName) {
        this(paramName, address:null);
    }
    Person(String name, String address) {
        this.name = name;
        this.address = address;
    }
    public String toString() {
        return "Person name: " + name + ", address: " +
            address + ", country: " +
            country;
    }
}
```

Analisis kegunaan kode program pada line 19-21.

- Kelas **Person** dirancang untuk memberikan representasi sederhana dari individu dengan informasi dasar seperti nama, alamat, dan negara.
- Konstruksi objek Person sangat fleksibel berkat overload pada konstruktor, yang memungkinkan pembuatan objek dengan berbagai kombinasi parameter.
- Metode `toString()` membuat debugging dan pencetakan informasi objek menjadi lebih mudah dengan menghasilkan output yang jelas dan informatif.

```
public class App {
    Run | Debug
    public static void main(String[] args) {
        Person person = new Person(name:"Abdullah",
        address:"Sitoluama");
        System.out.println(person);
    }
}
```

Analisis hasil dari kode program pada line 5.

- Pada **line 5**, pernyataan `System.out.println(person);` akan menghasilkan output yang informatif mengenai objek `person`, menunjukkan nama, alamat, dan negara.
- Hal ini memperlihatkan kemampuan Java untuk menyediakan representasi string dari objek, yang sangat berguna untuk debugging dan menampilkan informasi objek dengan cara yang mudah dibaca.

aa. Equals ()Method

```
class Person {
    final String country = "Indonesia";
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String toString() {
        return "Person name: " + name + ", age: " + age + ", country: " +
        country;
    }
    public boolean equals(Object o) {
        if (o == this) {
            return true;
        }
        if (!(o instanceof Person)) {
            return false;
        }
        Person person = (Person) o;
        if (this.age != person.age) {
            return false;
        }
        if (!this.name.equals(person.name)) {
            return false;
        }
        return true;
    }
}
```

Analisis kegunaan kode program pada line 15 – 36

- Kode program pada **line 15 hingga 36** mendefinisikan kelas `Person` yang memiliki kemampuan untuk menyimpan informasi dasar (nama, usia, negara) dan menyediakan metode untuk merepresentasikan objek sebagai string dan membandingkannya.
- Dengan adanya metode `toString()`, kita dapat dengan mudah mencetak informasi objek dengan cara yang mudah dibaca.
- Sementara itu, metode `equals()` meningkatkan kemampuan untuk membandingkan objek secara logis, sehingga membuat pengelolaan objek lebih intuitif, terutama saat digunakan dalam koleksi seperti **ArrayList**.

```

public class App {
    Run | Debug
    public static void main(String[] args) {
        Person p1 = new Person(name:"Abdullah", age:23);
        Person p2 = new Person(name:"Abdullah", age:23);
        Person p3 = new Person(name:"Abdullah", age:20);
        System.out.println(p1.equals(p2));
        System.out.println(p1.equals(p3));
    }
}

```

Analisis hasil kode program pada line 7 dan 8

- Pada **line 7**, hasilnya adalah **true** karena objek **p1** dan **p2** dianggap sama berdasarkan logika dalam metode **equals()**, yaitu mereka memiliki nama dan usia yang sama.
- Pada **line 8**, hasilnya adalah **false** karena meskipun kedua objek memiliki nama yang sama, usia yang berbeda menyebabkan metode **equals()** menganggap mereka tidak sama.
- Ini menunjukkan bahwa metode **equals()** telah berfungsi dengan baik dalam membandingkan objek berdasarkan atribut yang relevan.

bb. hashCode() Method

```

class Person {
    String name;
    int age;
    final String country = "Indonesia";
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String toString() {
        return "Person name: " + name + ", age: " + age + ", country: " +
    country;
    }
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + age;
        return result;
    }
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;
        Person person = (Person) o;
        if (age != person.age)
            return false;
        return name != null ? name.equals(person.name) : person.name == null;
    }
}

```

Analisis kegunaan kode program pada line 15 – 19.

- Kode pada **line 15 hingga 19** mendefinisikan metode **hashCode()** yang bertanggung jawab untuk menghasilkan nilai hash unik untuk objek Person berdasarkan atribut name dan age.
- Implementasi ini mengikuti praktik terbaik dalam

pengembangan perangkat lunak untuk mengurangi kemungkinan tabrakan nilai hash, dan menjaga konsistensi antara metode `equals()` dan `hashCode()`.

- Dengan cara ini, objek `Person` dapat digunakan dengan efektif dalam koleksi yang berbasis hash.

```
>...
public class App {
    Run | Debug
    public static void main(String[] args) {
        Person p1 = new Person(name:"Abdullah", age:23);
        Person p2 = new Person(name:"Abdullah", age:23);
        Person p3 = new Person(name:"Abdullah", age:20);
        System.out.println(p1.equals(p2));
        System.out.println(p1.hashCode() == p2.hashCode());
    }
}
```

Analisis hasil dari kode program pada line 8 dan 9.

- Pada **line 8**, hasilnya adalah `true` karena objek `p1` dan `p2` dianggap sama menurut logika dalam metode `equals()`.
- Pada **line 9**, hasilnya juga adalah `true` karena objek `p1` dan `p2` menghasilkan nilai hash yang sama menurut implementasi metode `hashCode()`.
- Ini menunjukkan bahwa implementasi metode `equals()` dan `hashCode()` bekerja dengan baik dan saling konsisten, mengikuti kontrak yang seharusnya.

cc. Final Case

```
public class SocialMedia {
    String name;
}
final class Facebook extends SocialMedia {
}
// class FakeFacebook extends Facebok{}
```

Analisis kegunaan kode program pada line 5. Analisis mengapa kode program pada line 8 mengalami error

- Kegunaan Line 5:** mendeklarasikan kelas `Facebook` sebagai kelas final yang tidak dapat diwarisi, dan mewarisi atribut dari kelas `SocialMedia`.
- Error pada Line 8:** adanya kesalahan pengetikan dalam nama kelas yang diturunkan, yaitu `Facebok`, yang tidak ada dalam konteks kode ini. Untuk memperbaikinya, kelas tersebut harus dirubah menjadi `Facebook`.

dd. Inner Class

```

public class Company {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public class Employee {
        private String name;
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
        public String getCompany() {
            return Company.this.name;
        }
    }
}

```

Analisis kegunaan kode program pada line 12 dan 24.

- o **Kegunaan Line 12:** Metode **getCompany()** dalam kelas **Employee** digunakan untuk mengakses dan mengembalikan nilai atribut **name** dari kelas induk **Company**. Ini memungkinkan objek **Employee** untuk mendapatkan informasi perusahaan yang relevan.
- o **Kegunaan Line 24:** Metode **setName(String name)** dalam kelas **Employee** digunakan untuk mengatur nilai atribut **name** dari objek **Employee**. Penggunaan kata kunci **this** membedakan antara parameter dan atribut, memungkinkan penetapan nilai dengan jelas dan terorganisir.

```

public class App {
    Run | Debug
    public static void main(String[] args) {
        Company company = new Company();
        company.setName(name:"IT DEL");
        Company.Employee employee = company.new Employee();
        employee.setName(name:"Abdullah");
        System.out.println(employee.getName());
        System.out.println(employee.getCompany());
        Company company2 = new Company();
        company2.setName(name:"BCA");
        Company.Employee employee2 = company2.new Employee();
        employee2.setName(name:"Kevin");
        System.out.println(employee2.getName());
        System.out.println(employee2.getCompany());
    }
}

```

Analisis kegunaan kode program pada line 7 dan 16.

Analisis hasil kode program pada line 11 dan 20.

ee. Final Method

```
public class SocialMedia {
    String name;
}
class Facebook extends SocialMedia {
final void login(String username, String password) {
// isi method
}
}
class FakeFacebook extends Facebook {
// void login(String username, String password) {
// // isi method
// }
```

Analisis kegunaan kode program pada line 6. Analisis mengapa kode program pada line 12 - 14 mengalami error.

- o **Kegunaan Line 6:** Kode pada **line 6** mendeklarasikan metode **login** yang bersifat final dalam kelas **Facebook**, sehingga tidak dapat diubah oleh subclass, memastikan konsistensi dalam perilaku metode tersebut.
- o **Error pada Line 12-14:** Kode pada **line 12-14** mengalami error karena berusaha untuk mengoverride metode **login** yang telah dideklarasikan sebagai **final** di kelas **Facebook**, yang tidak diperbolehkan dalam Java. Untuk menghindari error ini, metode **login** di **FakeFacebook** tidak boleh didefinisikan sama sekali.

ff. Anonymus Class

```
public interface Hello {
    void sayHello();
    void sayHello(String name);
}

public class App {
    Run | Debug
    public static void main(String[] args) {
        Hello hello = new Hello() {
            public void sayHello() {
                System.out.println("Hello");
            }
            public void sayHello(String name) {
                System.out.println("Hello " + name);
            }
        };
        hello.sayHello();
        hello.sayHello(name:"Abdullah");
    }
}
```

Analisis kegunaan kode program pada line 4 - 14. Analisis hasil dari kode program pada line 16 dan 17.

Program ini memanfaatkan *anonymous class* untuk mengimplementasikan metode dari kelas Hello tanpa perlu membuat kelas terpisah. Dengan cara ini, dua variasi dari metode sayHello diimplementasikan secara langsung, sehingga memungkinkan pemanggilan yang berbeda berdasarkan apakah parameter disertakan atau tidak. Hasil akhirnya adalah dua output yang dicetak ke konsol, yaitu "Hello" dan "Hello Abdullah".

gg. Static Keyword

```
package constant;
public class Constant1 {
    public static final String APP = "Bahasa Pemrograman
    Berorientasi
    Objek";
    public static final Integer VERSION = 1;
}
```

Analisis kegunaan kode program pada line 4 dan 5.

Baris 4 :

- **public static final:** Menunjukkan bahwa variabel ini bersifat *constant*, artinya nilainya tidak dapat diubah setelah diinisialisasi.
- **String:** Tipe data variabel APP adalah String, yang menyimpan teks.
- **APP:** Nama variabel untuk menyimpan informasi yang bersifat tetap, dalam hal ini teks yang menjelaskan "Bahasa Pemrograman Berorientasi Objek".
- **Nilai:** Nilai yang disimpan adalah "Bahasa Pemrograman Berorientasi Objek", yang dapat digunakan untuk menampilkan deskripsi singkat tentang aplikasi atau sistem.

Baris 5:

- **public static final:** Sama seperti baris sebelumnya, variabel ini bersifat konstan.
- **Integer:** Tipe data variabel VERSION adalah Integer, yang menyimpan angka bulat.
- **VERSION:** Nama variabel untuk menyimpan versi dari aplikasi atau sistem.
- **Nilai:** Nilai yang disimpan adalah 1, yang menandakan versi pertama dari aplikasi.

```
package constant;
public class Constant2 {
    public static final int PROCESSOR;
    static {
        PROCESSOR = Runtime.getRuntime().availableProcessors();
    }
}
```

Analisis kegunaan kode program pada line 4 dan 6 – 8.

Line 4 :

- **public static final:** Variabel ini dideklarasikan sebagai konstanta yang bersifat global dan hanya bisa diakses secara statis. Artinya, nilai variabel PROCESSOR tidak bisa

diubah setelah ditetapkan, dan dapat diakses langsung tanpa perlu membuat instance dari kelas Constant2.

- **int:** Tipe data yang digunakan adalah int (integer), yang menyimpan angka bulat.
- **PROCESSOR:** Variabel ini menyimpan jumlah prosesor yang tersedia di runtime.

Line 6-8 :

- **static {}:** Blok inisialisasi statis yang digunakan untuk menginisialisasi variabel statis. Blok ini akan dijalankan sekali ketika kelas Constant2 dimuat ke memori, sebelum ada instansiasi atau akses ke anggota statis kelas.
- **Runtime.getRuntime().availableProcessors():** Metode ini digunakan untuk mendapatkan jumlah prosesor yang tersedia pada mesin saat runtime. Nilainya akan diinisialisasi ke variabel PROCESSOR.

```
java > MathUtil > sum(int...)
public class MathUtil {
    public static int sum(int ... values) {
        int total = 0;
        for (var value : values) {
            total += value;
        }
        return total;
    }
}
```

Analisis kegunaan kode program pada line 3 - 9.

Line 3:

- **public:** Metode ini bisa diakses dari luar kelas MathUtil.
- **static:** Metode ini bersifat statis, sehingga bisa dipanggil tanpa perlu membuat objek dari kelas MathUtil.
- **int sum:** Metode ini mengembalikan nilai dengan tipe data int (bilangan bulat).
- **int... values:** Menunjukkan penggunaan varargs, yaitu memungkinkan metode ini menerima sejumlah argumen integer dalam bentuk array. Dengan varargs, pengguna bisa memanggil metode sum() dengan jumlah parameter int yang fleksibel (0 atau lebih).

Line 4 : Variabel total bertipe int diinisialisasi dengan nilai 0. Variabel ini akan digunakan untuk menyimpan hasil penjumlahan dari semua nilai yang diberikan dalam parameter values.

Line 5 - 7 :

- **for (var value : values):** Ini adalah bentuk perulangan *foreach* yang digunakan untuk menelusuri semua elemen dalam array values.
 - **var value:** Deklarasi tipe dinamis di mana value secara otomatis diinterpretasikan sebagai int, karena values adalah kumpulan elemen bertipe int.

- **total += value:** Setiap nilai dari values ditambahkan ke variabel total. Ini adalah cara untuk menghitung total penjumlahan semua nilai dalam array values.

Line 8 : Setelah seluruh elemen di dalam values dijumlahkan, nilai total dikembalikan sebagai hasil dari metode sum().

```
public class Country {  
    private String name;  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public static class City {  
        private String name;  
        public String getName() {  
            return name;  
        }  
        public void setName(String name) {  
            this.name = name;  
        }  
    }  
}
```

Analisis kegunaan kode program pada line 12.

Pada **line 12**, deklarasi public static class City menunjukkan bahwa kelas City adalah kelas bersarang statis di dalam kelas Country. Ini berarti kelas City dapat dibuat dan digunakan tanpa harus membuat instance dari kelas Country. Kegunaan dari pendekatan ini adalah untuk mengelompokkan logika yang berhubungan erat dengan kelas Country, namun tetap memberikan fleksibilitas dengan memisahkan ketergantungan antara kedua kelas. Kelas City menjadi entitas yang lebih mandiri, meskipun masih logis secara hierarki berada dalam kelas Country.

```

import static constant.Constant1.*;
import static constant.Constant2.PROCESSOR;
public class App {
Run|Debug
public static void main(String[] args) {
System.out.println(APP);
System.out.println(VERSION);
System.out.println(PROCESSOR);
System.out.println(Constant.phi);
int sum = MathUtil.sum(... values:1, 2, 3, 4, 5);
System.out.println(sum);
Country.City city = new Country.City();
city.setName(name:"Sitoluama");
System.out.println(city.getName());
}
}

```

Analisis tujuan kode program pada line 1, 2, 12 dan 15.

Analisis hasil dari kode program pada line 7, 8, 9, 10, 13 dan 17.

Line 1 : Kode ini melakukan import statis dari seluruh anggota di dalam kelas constant.Constant1. Dengan kata lain, Anda dapat menggunakan semua konstanta yang didefinisikan dalam Constant1 tanpa menyebutkan nama kelas secara eksplisit.

Line 2 : Import statis hanya untuk konstanta PROCESSOR dari kelas constant.Constant2. Ini memungkinkan konstanta PROCESSOR digunakan di dalam kode tanpa harus memanggilnya menggunakan nama lengkap kelas (Constant2.PROCESSOR).

Line 12 : Baris ini memanggil metode sum() dari kelas MathUtil untuk menghitung penjumlahan dari angka-angka 1, 2, 3, 4, 5. Hasil dari operasi ini disimpan dalam variabel sum.

Line 15 : Baris ini membuat objek baru dari kelas bersarang City yang ada di dalam kelas Country. Objek city digunakan untuk mengakses metode atau properti yang dimiliki oleh kelas City.

Line 7 : Baris ini akan mencetak nilai dari konstanta APP, yang kemungkinan besar telah diimpor dari kelas Constant1 (karena tidak ada deklarasi langsung).

Line 8 : Akan mencetak nilai dari konstanta VERSION, yang juga mungkin diimpor dari Constant1.

Line 9 : Akan mencetak nilai dari konstanta PROCESSOR, yang berasal dari Constant2 (karena diimpor secara eksplisit pada baris 2)

Line 10 : Mencetak nilai dari phi, sebuah properti atau konstanta di dalam kelas Constant. Namun, nilai ini tidak diimpor secara statis sehingga harus dipanggil dengan Constant.phi.

Line 13 : Baris ini akan mencetak hasil dari penjumlahan 1, 2, 3, 4, 5, yang dihitung pada baris 12. Hasilnya adalah 15

Line 17 : Mencetak nama dari objek city, yang sebelumnya di-set menjadi "Sitoluama" pada baris 16. Jadi output yang dihasilkan adalah "Sitoluama".

hh. Record Class

```
public record LoginRequest(String username, String password) {
    public LoginRequest {
        System.out.println("Membuat object LoginRequest");
    }
    public LoginRequest(String username) {
        this(username, password:"");
    }
    public LoginRequest() {
        this(username:"", password(""));
    }
}
```

Analisis kegunaan kode program pada line 1, 3-5, 8, dan 12.

- Line 1 : Ini adalah deklarasi dari sebuah **record** di Java, yang merupakan tipe data khusus untuk menyimpan data. Dalam hal ini, LoginRequest adalah record yang menyimpan dua field: username dan password. Record di Java secara otomatis menghasilkan constructor, getter untuk field, equals(), hashCode(), dan toString(). Tujuan utama dari line ini adalah mendefinisikan data yang akan dipegang oleh LoginRequest tanpa perlu menuliskan semua metode boilerplate secara manual.
- Line 3-5 : **custom canonical constructor** dari record. Meskipun Java secara otomatis membuat constructor untuk record, kamu dapat menambahkan behavior tertentu ketika objek dibuat. Di sini, pada saat objek LoginRequest dibuat, pesan "Membuat object LoginRequest" akan dicetak ke konsol. Fungsinya adalah untuk menambah log atau pemberitahuan saat objek diinisialisasi, yang bisa berguna untuk debugging atau tracking pembuatan objek.
- Line 8 : Ini adalah **constructor tambahan** yang menerima hanya username sebagai parameter. Jika pengguna hanya memberikan username, maka constructor ini akan dipanggil, dan password secara otomatis di-set menjadi string kosong (""). Constructor ini memberikan fleksibilitas dalam pembuatan objek LoginRequest tanpa memerlukan password.
- Line 12 : **constructor tanpa argumen (no-argument constructor)** yang menetapkan nilai default untuk username dan password menjadi string kosong (""). Fungsinya adalah untuk memungkinkan pembuatan objek LoginRequest tanpa memberikan nilai apa pun, dengan asumsi bahwa pengguna ingin membuat objek dengan data kosong atau data default.

ii. Enum Class

```
'...
public enum Level {
    STANDARD(description:"Standard Level"),
    PREMIUM(description:"Premium Level"),
    VIP(description:"VIP Level");
    private String description;
    Level(String description) {
        this.description = description;
    }
    public String getDescription() {
        return description;
    }
}
```

Analisis kode program pada line 1-15.

Kode ini mendefinisikan enum Level dengan tiga konstanta: STANDARD, PREMIUM, dan VIP. Setiap konstanta memiliki deskripsi yang diberikan melalui konstruktor. Variabel description diinisialisasi melalui konstruktor dan dapat diambil dengan menggunakan metode getter getDescription().

Kegunaan utama dari kode ini adalah:

- Membatasi pilihan level menjadi hanya tiga nilai (STANDARD, PREMIUM, VIP).
- Menyediakan cara untuk memberikan deskripsi tambahan untuk setiap level.
- Menerapkan encapsulation untuk mengendalikan akses ke properti deskripsi dari luar enum.

```
public class Customer {
    private String name;
    private Level level;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Level getLevel() {
        return level;
    }
    public void setLevel(Level level) {
        this.level = level;
    }
}
```

Analisis kode program pada line 3 dan 17.

- **Line 3** mendeklarasikan variabel level yang akan menyimpan status atau tingkatan pelanggan menggunakan enum Level (dengan nilai seperti STANDARD, PREMIUM, atau VIP).
- **Line 17** mendeklarasikan setter untuk variabel level, yang memungkinkan program atau pengguna eksternal mengubah tingkatan pelanggan.

11S2324-PBO

Laporan Praktikum

```
public class App {  
    Run | Debug  
    public static void main(String[] args) {  
        Customer customer = new Customer();  
        customer.setName(name:"Abdullah");  
        customer.setLevel(Level.VIP);  
        System.out.println(customer.getName());  
        System.out.println(customer.getLevel());  
        System.out.println(customer.getLevel().getDescription());  
        // Konversi Enum ke String  
        Level level = Level.valueOf("VIP");  
        System.out.println(level);  
        System.out.println(Level.STANDARD.name());  
        System.out.println("Print Level:");  
        for (var value : Level.values()) {  
            System.out.println("- " + value);  
        }  
    }  
}
```

Analisis kode program pada line 6, 9, 10, 13, 15, 18 - 20.

- **Line 6** menginisialisasi objek Customer.
- **Line 9** dan **10** menggunakan setter untuk menetapkan nama pelanggan menjadi "Abdullah" dan levelnya menjadi VIP.
- **Line 13** menampilkan deskripsi level pelanggan (dalam hal ini, "VIP Level") dengan menggunakan getter chaining.
- **Line 15** mengonversi string "VIP" menjadi nilai enum Level.VIP.
- **Line 18-20** mencetak semua nilai enum Level menggunakan loop, menampilkan semua level yang ada (STANDARD, PREMIUM, VIP).

2. Mengimplementasikan Class Diagram

a) Class Name, Attribute dan Method

Hak Akses	Public (+)	Package (~)	Protected (#)	Private (-)
Anggota dari kelas yang sama	Y	Y	Y	Y
Anggota kelas turunan	Y	Y	Y	N
Anggota dari kelas lain	Y	dalam paket yang sama	N	N

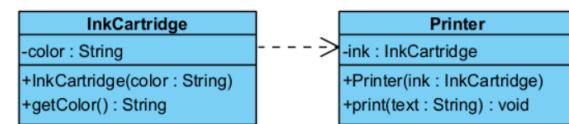
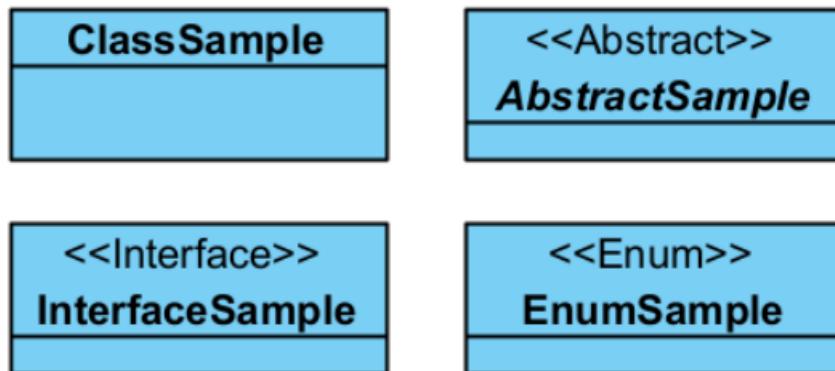


Diagram ini menunjukkan bagaimana **komposisi** antara dua kelas dapat digunakan untuk mendefinisikan hubungan yang erat antara objek (printer dan cartridge tinta). Program yang menyertainya mengikuti konsep **OOP** seperti **encapsulation**, **constructor**, dan **method invocation** untuk membangun sistem sederhana yang mensimulasikan printer yang menggunakan cartridge tinta untuk mencetak teks.

```

public class Person {
    public String name;
    protected String gender;
    int age; // access modifier: default atau package
    private String address;
    public Person() {
    }
    public Person(String name, String gender, int age, String address) {
        this.name = name;
        this.gender = gender;
        this.age = age;
        this.address = address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    public String getAddress() {
        return this.address;
    }
}
  
```

b) Class, Class Abstract, Interface dan Enum



```

public class ClassSample {
}

public class AbstractSample {

}

public class InterfaceSample {

}

public class EnumSample {
}
  
```

- **Class:** Kelas umum yang dapat diinstansiasi.
- **Abstract Class:** Kelas yang tidak dapat diinstansiasi dan dapat memiliki metode abstrak yang harus diimplementasikan oleh subclass.
- **Interface:** Mendefinisikan kontrak atau metode tanpa implementasi yang harus diimplementasikan oleh kelas lain.
- **Enum:** Mendefinisikan kumpulan konstanta yang tetap dan tidak berubah.

c) Realisasi



Diagram tersebut menunjukkan hubungan antara antarmuka **Animal** dan kelas **Monkey**. **Animal** mendeklarasikan metode `walk()` yang mengembalikan **String**, tanpa implementasi. Kelas **Monkey** mengimplementasikan antarmuka ini, sehingga harus menyediakan definisi konkret dari metode `walk()`. Diagram ini menggambarkan konsep **interface** dan **polimorfisme** dalam OOP, di mana kelas **Monkey** mengikuti kontrak yang ditetapkan oleh **Animal**, namun dengan implementasi spesifik untuk metode `walk()`.

d) Inheritance (Generalisasi)

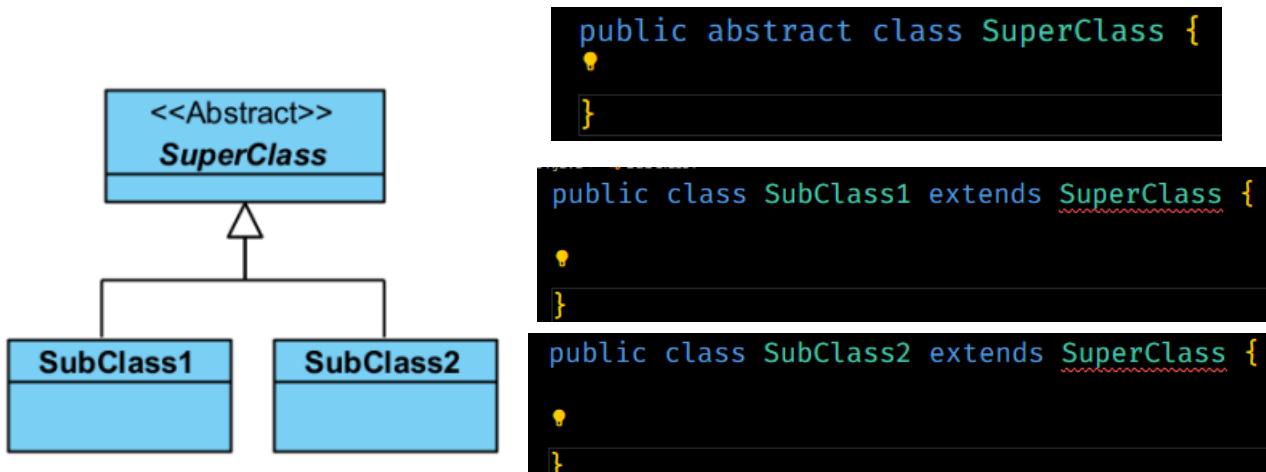


Diagram tersebut menggambarkan konsep **inheritance** dalam **Object-Oriented Programming (OOP)**. Diagram ini terdiri dari kelas abstrak bernama **SuperClass**, yang tidak dapat diinstansiasi secara langsung. Kelas ini berfungsi sebagai dasar untuk kelas turunan, yaitu **SubClass1** dan **SubClass2**, yang mewarisi atribut dan metode dari **SuperClass**. Karena **SuperClass** bersifat abstrak, metode atau atribut di dalamnya mungkin tidak memiliki implementasi penuh, dan kelas turunannya, **SubClass1** serta **SubClass2**, harus mengimplementasikan atau memperluas metode tersebut. Ini menggambarkan pewarisan dan penggunaan kelas abstrak dalam OOP, di mana kelas-kelas turunan mewarisi perilaku dasar dari kelas induk dan menyesuaikan dengan kebutuhan spesifik mereka.

e) Association

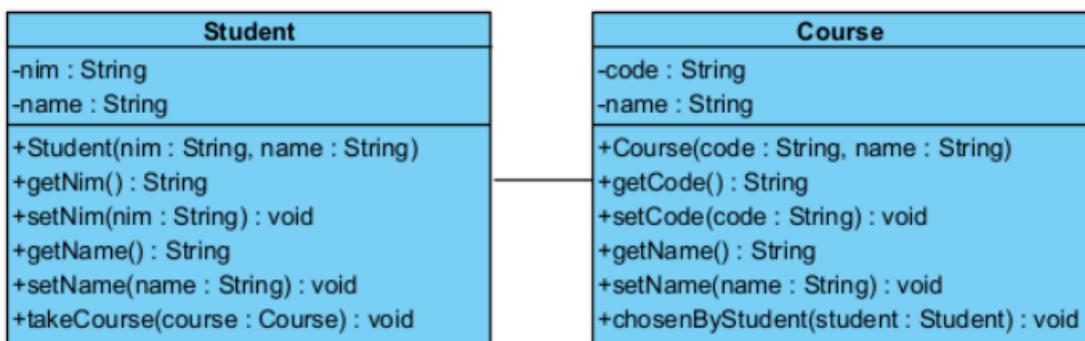


Diagram kelas ini mewakili sebuah sistem sederhana yang berhubungan dengan data mahasiswa dan mata kuliah. Kelas **Student** merepresentasikan seorang mahasiswa dengan atribut seperti nomor induk dan nama. Kelas **Course** merepresentasikan suatu mata kuliah dengan kode dan nama.

Hubungan antara kedua kelas ini menunjukkan bahwa seorang mahasiswa dapat mengambil beberapa mata kuliah, dan sebaliknya, suatu mata kuliah dapat dipilih oleh banyak mahasiswa. Ini adalah hubungan *many-to-many*.

```
public class Student {  
    private String nim;  
    private String name;  
    public Student(String nim, String name) {  
        this.nim = nim;  
        this.name = name;  
    }  
    public String getNim() {  
        return this.nim;  
    }  
    public void setNim(String nim) {  
        this.nim = nim;  
    }  
    public String getName() {  
        return this.name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    // Method asosiasi untuk mengaitkan Course dengan  
    kelas Student  
    public void takeCourse(Course course) {  
        System.out.println("Mata kuliah " + course.getName()  
        + " dipilih oleh "  
        + nam  
    }  
}
```

```
public class Course {  
    private String code;  
    private String name;  
    public Course(String code, String name) {  
        this.code = code;  
        this.name = name;  
    }  
    public String getCode() {  
        return this.code;  
    }  
    public void setCode(String code) {  
        this.code = code;  
    }  
    public String getName() {  
        return this.name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    // Method asosiasi untuk mengaitkan Student dengan  
    kelas Course  
    public void chosenByStudent(Student student) {  
        System.out.println(name + " mengambil mata kuliah " +  
        student.getName());  
    }  
}
```

f) Aggregation

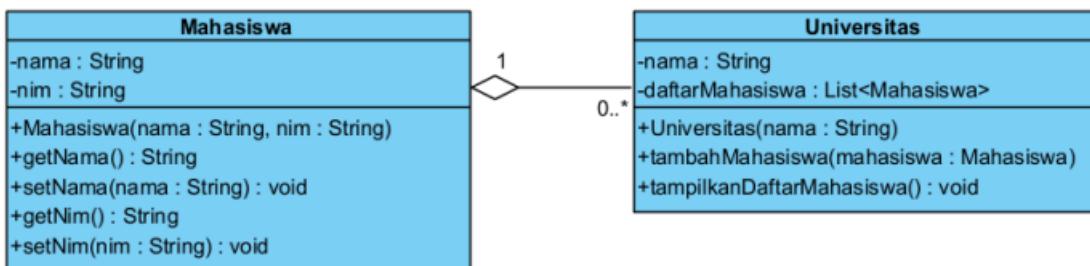


Diagram kelas ini memberikan gambaran yang jelas tentang hubungan antara entitas Mahasiswa dan Universitas dalam sebuah sistem informasi akademik. Dengan memahami diagram ini, kita dapat membangun program yang dapat mengelola data mahasiswa dengan baik.

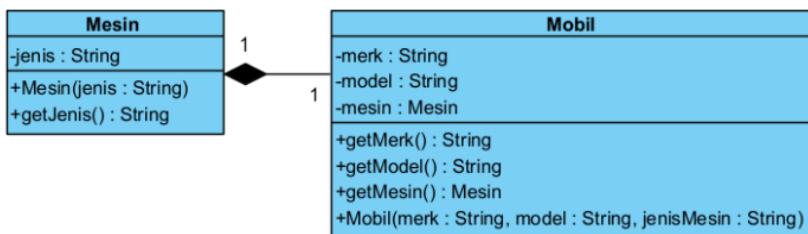
```

public class Mahasiswa {
    private String nama;
    private String nim;
    public Mahasiswa(String nama, String nim) {
        this.nama = nama;
        this.nim = nim;
    }
    public String getNama() {
        return this.nama;
    }
    public void setNama(String nama) {
        this.nama = nama;
    }
    public String getNim() {
        return this.nim;
    }
    public void setNim(String nim) {
        this.nim = nim;
    }
}
  
```

```

import java.util.List;
import java.util.ArrayList;
public class Universitas {
    private String nama;
    private List<Mahasiswa> daftarMahasiswa;
    public Universitas(String nama) {
        this.nama = nama;
        this.daftarMahasiswa = new ArrayList<>();
    }
    public void tambahMahasiswa(Mahasiswa mahasiswa) {
        daftarMahasiswa.add(mahasiswa);
    }
    public void tampilkanDaftarMahasiswa() {
        System.out.println("Daftar Mahasiswa di Universitas");
        " + nama +
        ":" );
        for (Mahasiswa mahasiswa : daftarMahasiswa) {
            System.out.println("Nama: " + mahasiswa.getNama() +
            ", NIM: " +
            mahasiswa.getNim());
        }
    }
}
  
```

g) Composition



```

public class Mesin {
    private String jenis;
    public Mesin(String jenis) {
        this.jenis = jenis;
    }
    public String getJenis() {
        return jenis;
    }
}
  
```

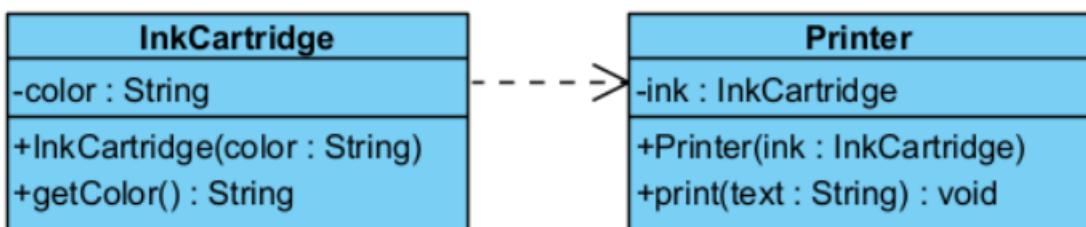
```

public class Mobil {
    private String merk;
    private String model;
    private Mesin mesin;
    public Mobil(String merk, String model, String jenisMesin) {
        this.merk = merk;
        this.model = model;
        this.mesin = new Mesin(jenisMesin);
    }
    public String getMerk() {
        return merk;
    }
    public String getModel() {
        return model;
    }
    public Mesin getMesin() {
        return mesin;
    }
}

```

Diagram di atas menunjukkan hubungan komposisi antara dua kelas, yaitu **Mobil** dan **Mesin**, dalam pemrograman berorientasi objek. Dalam hubungan ini, setiap objek **Mobil** terdiri dari satu objek **Mesin**, yang digambarkan melalui simbol komposisi (diamond hitam). Kelas **Mesin** memiliki atribut jenis yang menunjukkan tipe mesin dan menyediakan metode untuk mengambil jenis mesin tersebut. Di sisi lain, kelas **Mobil** memiliki atribut merk, model, serta mesin, yang merujuk pada objek dari kelas **Mesin**. **Mobil** juga menyediakan metode untuk mengambil merek, model, dan informasi tentang mesin yang dimilikinya. Kelas **Mobil** tidak bisa ada tanpa kelas **Mesin**, karena setiap objek **Mobil** bergantung pada objek **Mesin** yang dimilikinya. Kesimpulannya, diagram ini merepresentasikan hubungan yang erat antara mobil dan mesinnya, di mana mobil tidak dapat dipisahkan dari mesinnya, mencerminkan prinsip **komposisi** dalam pemrograman berorientasi objek.

h) Dependency



Hubungan antara **InkCartridge** dan **Printer** adalah hubungan *composition* atau *aggregation* (ditunjukkan oleh garis putus-putus dengan panah berlian), di mana printer membutuhkan sebuah cartridge tinta agar bisa berfungsi, tetapi cartridge bisa berdiri sendiri tanpa printer.

Implementasi Program: Program ini akan memodelkan sebuah printer yang memerlukan cartridge tinta dengan warna tertentu. Ketika printer mencetak sesuatu, ia akan menggunakan cartridge tinta untuk mencetak teks. Sebagai contoh, objek *InkCartridge* akan diinisialisasi dengan warna, kemudian objek *Printer* akan dibuat dengan memasukkan objek *InkCartridge* tersebut. Printer kemudian

11S2324-PBO

Laporan Praktikum

dapat mencetak teks menggunakan warna dari *InkCartridge* yang tersedia.

```
public class InkCartridge {  
    private String color;  
  
    public InkCartridge(String color) {  
        this.color = color;  
    }  
  
    public String getColor() {  
        return color;  
    }  
}
```

```
public class Printer {  
    private InkCartridge ink;  
    public Printer(InkCartridge ink) {  
        this.ink = ink;  
    }  
    public void print(String text) {  
        System.out.println("Printing: " + text);  
        System.out.println("Using ink color: " + ink.getColor());  
    }  
}
```

3. Studi Kasus 1: Student Information

```

import java.util.Scanner;

public class App {
    Run|Debug
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input data mahasiswa
        // System.out.print("Masukkan NIM: ");
        String studentNIM = sc.nextLine();
        // System.out.print("Masukkan Nama: ");
        String studentName = sc.nextLine();
        // System.out.print("Masukkan Umur: ");
        int studentAge = Integer.parseInt(sc.nextLine());

        // Input data rumah dan alamat
        // System.out.print("Masukkan Model Rumah: ");
        String homeModel = sc.nextLine();
        // System.out.print("Masukkan Kota Alamat: ");
        String addressCity = sc.nextLine();
        // System.out.print("Masukkan Jalan Alamat: ");
        String addressStreet = sc.nextLine();

        // Input data kursus
        // System.out.print("Masukkan Nama Kursus: ");
        String courseName = sc.nextLine();
        // System.out.print("Masukkan Jumlah SKS Kursus: ");
        int courseCredits = Integer.parseInt(sc.nextLine());

        // Membuat objek Student, Home, dan Course
        Person person = new Student(studentNIM, studentName, studentAge);
        Home home = new Home(homeModel, person, addressCity, addressStreet);
        Course course = new Course(courseName, courseCredits);

        // Enroll course jika person adalah student
        if (person instanceof Student) {
            Student student = (Student) person;
            student.enrollCourse(course);
        }

        // Menampilkan informasi
        System.out.println("Person Name: " + person.getName());
        System.out.println("Person Age: " + person.getAge());

        if (person instanceof Student) {
            Student student = (Student) person;
            System.out.println("Student ID: " + student.getNim());
            System.out.println("Student Home Model: " + home.getModel());
            System.out.println("Student Address: " +
                home.getAddress().getStreet() + ", " + home.getAddress().getCity());

            // Menangani kemungkinan course belum di-enroll
            Course enrolledCourse = student.getEnrolledCourse();
            if (enrolledCourse != null) {
                System.out.println("Enrolled Course: " +
                    enrolledCourse.getName() + "[" + enrolledCourse.getCredits() + "skls]");
            } else {
                System.out.println("No course enrolled yet.");
            }
            student.work();
        }

        sc.close();
    }
}

```

Penjelasan :

1. Mengimpor Scanner

Kode `import java.util.Scanner;` digunakan untuk mengimpor kelas `Scanner` dari pustaka Java, yang digunakan untuk membaca input dari pengguna melalui konsol.

2. Deklarasi Kelas

Kode `public class App {` mendeklarasikan kelas `App` yang menjadi kelas utama tempat eksekusi program berlangsung.

3. Fungsi Main

Kode `public static void main(String[] args) {` mendefinisikan metode `main`, yang merupakan titik masuk dari eksekusi program Java.

4. Membuat Objek Scanner

Kode `Scanner sc = new Scanner(System.in);` mendefinisikan objek `Scanner` bernama `sc`, yang digunakan untuk mengambil input dari pengguna melalui konsol.

11S2324-PBO

Laporan Praktikum

5. Membaca Input NIM

`String studentNIM = sc.nextLine();` membaca input NIM dari pengguna dan menyimpannya dalam variabel `studentNIM`.

6. Membaca Input Nama Mahasiswa

`String studentName = sc.nextLine();` membaca input nama dari pengguna dan menyimpannya dalam variabel `studentName`.

7. Membaca Input Umur Mahasiswa

`int studentAge = Integer.parseInt(sc.nextLine());` membaca input umur dari pengguna sebagai string, kemudian mengonversinya menjadi integer dan menyimpannya di variabel `studentAge`.

8. Membaca Input Model Rumah

`String homeModel = sc.nextLine();` membaca input model rumah dari pengguna dan menyimpannya dalam variabel `homeModel`.

9. Membaca Input Kota Alamat

`String addressCity = sc.nextLine();` membaca input kota dari pengguna dan menyimpannya dalam variabel `addressCity`.

10. Membaca Input Jalan Alamat

`String addressStreet = sc.nextLine();` membaca input jalan dari pengguna dan menyimpannya dalam variabel `addressStreet`.

11. Membaca Input Nama Kursus

`String courseName = sc.nextLine();` membaca input nama kursus dari pengguna dan menyimpannya dalam variabel `courseName`.

12. Membaca Input SKS Kursus

`int courseCredits = Integer.parseInt(sc.nextLine());` membaca input SKS kursus dari pengguna sebagai string, kemudian mengonversinya menjadi integer dan menyimpannya dalam variabel `courseCredits`.

13. Membuat Objek Student

`Person person = new Student(studentNIM, studentName, studentAge);` membuat objek `Student` dengan parameter NIM, nama, dan umur. Objek ini disimpan dalam variabel `person`, yang bertipe `Person`.

14. Membuat Objek Home

`Home home = new Home(homeModel, person, addressCity, addressStreet);` membuat objek `Home` dengan model rumah, pemilik (`person`), kota, dan jalan alamat.

15. Membuat Objek Course

`Course course = new Course(courseName, courseCredits);` membuat objek `Course` dengan nama kursus dan jumlah SKS.

16. Pengecekan jika Person adalah Student

```
if (person instanceof Student) { mengecek apakah objek person adalah instance dari kelas Student menggunakan instanceof.
```

17. Mendaftarkan Kursus

`Student student = (Student) person;` melakukan downcast dari `Person` ke `Student` agar bisa mengakses metode dan properti `Student`.

`student.enrollCourse(course);` memanggil metode `enrollCourse` untuk mendaftarkan kursus pada mahasiswa.

18. Menampilkan Nama dan Umur Person

`System.out.println("Person Name: " + person.getName());` menampilkan nama person menggunakan metode `getName()`.

`System.out.println("Person Age: " + person.getAge());` menampilkan umur person menggunakan metode `getAge()`.

19. Menampilkan NIM dan Informasi Rumah

`System.out.println("Student ID: " + student.getNim());` menampilkan NIM mahasiswa.

```
System.out.println("Student Home Model: " + home.getModel()); menampilkan
model rumah mahasiswa.
System.out.println("Student Address: " + home.getAddress().getStreet() + ",
" + home.getAddress().getCity()); menampilkan alamat lengkap mahasiswa.
```

20. Pengecekan Kursus yang Diambil

```
Course enrolledCourse = student.getEnrolledCourse(); mendapatkan kursus yang
telah diambil oleh mahasiswa.
if (enrolledCourse != null) { mengecek apakah kursus telah diambil. Jika sudah,
tampilkan kursus tersebut.
```

21. Menampilkan Informasi Kursus

```
System.out.println("Enrolled Course: " + enrolledCourse.getName() + " [" +
enrolledCourse.getCredits() + " sks]"); menampilkan nama dan jumlah SKS dari
kursus yang telah diambil mahasiswa.
```

22. Jika Belum Ada Kursus yang Diambil

```
System.out.println("No course enrolled yet."); menampilkan pesan jika mahasiswa
belum mengambil kursus.
```

23. Memanggil Metode Work pada Student

```
student.work(); memanggil metode work() pada objek student, yang akan menampilkan
bahwa mahasiswa sedang belajar.
```

24. Menutup Scanner

```
sc.close(); menutup objek Scanner setelah semua input selesai diambil.
```

```
public class Address {
    private String city;
    private String street;

    protected Address(String city , String street){
        this.city = city;
        this.street = street;
    }

    protected String getCity(){
        return city;
    }

    protected String getStreet(){
        return street;
    }
}
```

Penjelasan :

- Deklarasi kelas Address yang bersifat public:** Kelas ini bernama Address dan dapat diakses dari luar kelas. Kata kunci public menunjukkan bahwa kelas ini bisa diakses oleh kelas lain di dalam dan di luar paket yang sama.

- **Deklarasi variabel private String city:** Ini adalah deklarasi variabel instance `city` dengan tipe data `String`. Kata kunci `private` berarti variabel ini hanya bisa diakses dari dalam kelas `Address` saja. Kelas lain tidak bisa mengakses variabel ini secara langsung.
- **Deklarasi variabel private String street:** Sama seperti variabel `city`, `street` juga merupakan variabel instance dengan tipe data `String` yang bersifat `private`, artinya hanya bisa diakses dari dalam kelas `Address` saja.
- **Deklarasi konstruktor protected Address(String city, String street):** Konstruktor ini digunakan untuk membuat objek dari kelas `Address`. Kata kunci `protected` berarti konstruktor ini hanya bisa diakses oleh kelas itu sendiri, subclass-nya, atau kelas lain di dalam paket yang sama. Konstruktor menerima dua parameter, yaitu `city` dan `street` yang bertipe `String`.
- **this.city = city;:** Ini adalah pernyataan yang digunakan untuk menginisialisasi variabel instance `city` dengan nilai dari parameter `city`. Kata kunci `this` merujuk pada objek saat ini dari kelas `Address`.
- **this.street = street;:** Ini adalah pernyataan yang digunakan untuk menginisialisasi variabel instance `street` dengan nilai dari parameter `street`. Sama seperti `city`, kata kunci `this` merujuk pada objek saat ini.
- **Deklarasi metode protected String getCity():** Metode ini bertipe `String` dan digunakan untuk mengembalikan nilai variabel `city`. Kata kunci `protected` berarti metode ini bisa diakses dari dalam kelas, subclass-nya, atau kelas lain di dalam paket yang sama.
- **return city;:** Pernyataan ini mengembalikan nilai dari variabel `city` kepada pemanggil metode `getCity()`.
- **Deklarasi metode protected String getStreet():** Metode ini juga bertipe `String` dan digunakan untuk mengembalikan nilai variabel `street`. Sama seperti metode `getCity()`, metode ini bersifat `protected`.
- **return street;:** Pernyataan ini mengembalikan nilai dari variabel `street` kepada pemanggil metode `getStreet()`.

```

public class Course {
    String name;
    int credits;

    protected Course(String name , int credit
        |   this.name = name;
        |   this.credits = credits;
    }

    protected String getName(){
        |   return name;
    }

    protected int getCredits(){
        |   return credits;
    }
}

```

Penjelasan :

- **Deklarasi kelas Course yang bersifat public:** Kelas ini bernama Course dan dapat diakses dari mana saja.
- **Deklarasi variabel instance name bertipe String:** Variabel ini menyimpan nama dari course. Tidak ada akses modifier (default), sehingga hanya bisa diakses oleh kelas dalam paket yang sama.
- **Deklarasi variabel instance credits bertipe int:** Variabel ini menyimpan jumlah kredit dari course. Sama seperti name, aksesnya default.
- **Deklarasi konstruktor protected Course(String name, int credits):** Konstruktor ini akan dipanggil ketika objek Course dibuat, menerima parameter name (nama course) dan credits (jumlah kredit). Kata kunci protected berarti konstruktor ini hanya bisa diakses dari subclass atau kelas lain di dalam paket yang sama.
- **Inisialisasi variabel instance name dengan parameter name:** this.name = name; digunakan untuk mengisi variabel instance name dengan nilai yang diberikan pada parameter name.
- **Inisialisasi variabel instance credits dengan parameter credits:** this.credits = credits; digunakan untuk mengisi variabel instance credits dengan nilai yang diberikan pada parameter credits.
- **Deklarasi metode protected getName() bertipe String:** Metode ini mengembalikan nilai dari variabel instance name. Kata kunci protected berarti metode ini bisa diakses dari subclass atau kelas lain di dalam paket yang sama.
- **Metode getName() mengembalikan nilai name:** return name; digunakan untuk mengembalikan nilai name saat metode ini dipanggil.
- **Deklarasi metode protected getCredits() bertipe int:** Metode ini mengembalikan nilai dari variabel instance credits. Kata kunci

protected membuat metode ini dapat diakses oleh subclass atau kelas lain di paket yang sama.

- **Metode getCredits() mengembalikan nilai credits:** return credits; digunakan untuk mengembalikan nilai credits saat metode ini dipanggil.

11S2324-PBO

Laporan Praktikum

```

public class Home {
    private String model;
    private Address address;
    private Person owner;

    protected Home(String model , Person owner ,String addressCity ,String addressStreet){
        this.model = model;
        this.address = new Address(addressCity,addressStreet);
        this.owner = owner;

    }

    protected String getModel(){
        return model;
    }

    protected Address getAddress(){
        return address;
    }

    protected Person getOwner(){
        return owner;
    }
}

```

Penjelasan :

- **Mendefinisikan Kelas Home**

public class Home { mendeklarasikan kelas Home yang bersifat public, artinya dapat diakses dari kelas lain dalam program.

- **Deklarasi Variabel model**

private String model; mendefinisikan variabel instance model yang bersifat private. Variabel ini digunakan untuk menyimpan informasi tentang model rumah, dan hanya dapat diakses dari dalam kelas Home.

- **Deklarasi Variabel address**

private Address address; mendeklarasikan variabel address yang bersifat private, digunakan untuk menyimpan objek Address yang merepresentasikan alamat rumah.

- **Deklarasi Variabel owner**

private Person owner; mendeklarasikan variabel owner yang bersifat private, yang menyimpan objek Person sebagai pemilik rumah.

- **Konstruktur Kelas Home**

protected Home(String model, Person owner, String addressCity, String addressStreet) { mendefinisikan konstruktor yang bersifat protected. Konstruktor ini digunakan untuk menginisialisasi objek Home dengan model rumah, pemilik (Person), dan alamat (kota dan jalan).

- **Inisialisasi Variabel model**

this.model = model; menetapkan nilai dari parameter model yang diterima oleh konstruktor ke variabel instance model.

- **Membuat Objek Address**

this.address = new Address(addressCity, addressStreet); membuat objek baru dari kelas Address menggunakan kota dan jalan yang diberikan, dan menyimpannya di variabel instance address.

- **Inisialisasi Variabel owner**

this.owner = owner; menetapkan nilai dari parameter owner ke variabel instance owner, yang merupakan objek Person.

- **Metode getModel**

`protected String getModel() { mendefinisikan metode getModel yang bersifat protected. Metode ini mengembalikan nilai dari variabel model.`

- **Metode getAddress**

`protected Address getAddress() { mendefinisikan metode getAddress yang bersifat protected. Metode ini mengembalikan objek Address yang disimpan di variabel address.`

- **Metode getOwner**

`protected Person getOwner() { mendefinisikan metode getOwner yang bersifat protected. Metode ini mengembalikan objek Person yang merepresentasikan pemilik rumah.`

```
public abstract class Person {
    String name;
    int age;

    protected Person(String name ,int age){
        this.name = name;
        this.age = age;
    }

    protected String getName() {
        return name; // Implement getter
    }

    protected int getAge() {
        return age; // Implement getter
    }

    public abstract void work();
}
```

Penjelasan :

- **Mendeklarasikan Kelas Abstrak Person**

`public abstract class Person { mendeklarasikan kelas Person sebagai kelas abstrak. Kelas abstrak berarti tidak dapat diinstansiasi secara langsung, dan dapat berisi metode abstrak yang harus diimplementasikan oleh kelas turunan.`

- **Deklarasi Variabel name**

`String name; mendefinisikan variabel instance name yang bersifat default (tidak ada modifier), sehingga hanya dapat diakses dalam paket yang sama. Variabel ini menyimpan nama dari objek Person.`

- **Deklarasi Variabel age**

`int age; mendefinisikan variabel instance age, yang menyimpan umur dari objek Person. Aksesnya juga bersifat default.`

- **Konstruktor Kelas Person**

`protected Person(String name, int age) { mendefinisikan konstruktor yang bersifat`

protected, yang artinya hanya dapat diakses oleh kelas turunan atau kelas dalam paket yang sama. Konstruktor ini digunakan untuk menginisialisasi variabel `name` dan `age` saat objek `Person` dibuat.

- **Inisialisasi Variabel name**

`this.name = name;` menetapkan nilai dari parameter `name` yang diterima konstruktor ke variabel instance `name`.

- **Inisialisasi Variabel age**

`this.age = age;` menetapkan nilai dari parameter `age` yang diterima konstruktor ke variabel instance `age`.

- **Metode getName**

`protected String getName() {` mendefinisikan metode `getName` yang bersifat `protected`. Metode ini mengembalikan nilai `name` dari objek `Person`. Metode ini hanya dapat diakses oleh kelas turunan atau kelas dalam paket yang sama.

- **Metode getAge**

`protected int getAge() {` mendefinisikan metode `getAge` yang bersifat `protected`. Metode ini mengembalikan nilai `age` dari objek `Person`.

- **Deklarasi Metode Abstrak work**

`public abstract void work();` mendefinisikan metode abstrak `work()`. Metode ini tidak memiliki implementasi di kelas `Person` dan harus diimplementasikan oleh kelas turunan. Aksesnya bersifat `public`, sehingga metode ini harus diimplementasikan dengan akses publik pada kelas turunan.

```

public class Student extends Person implements Workable {
    private String nim;
    private Course enrollCourse;

    // Constructor untuk Student
    protected Student(String nim, String name, int age) {
        super(name, age); // Memanggil constructor dari superclass
        this.nim = nim;
    }

    // Getter untuk nim
    protected String getNim(){
        return nim;
    }

    // Getter untuk course yang di-enroll
    protected Course getEnrolledCourse(){
        return enrollCourse;
    }

    // Method untuk enroll course
    protected void enrollCourse(Course course){
        this.enrollCourse = course; // Set course yang di-enroll
    }

    // Override method work dari class Person
    @Override
    public void work(){
        System.out.println("Student is studying");
    }
}
  
```

Penjelasan :

- **Deklarasi kelas Student yang extends dari Person dan implements Workable:** Kelas Student merupakan subclass dari kelas Person dan juga mengimplementasikan interface Workable. Ini artinya, Student mewarisi sifat-sifat dari Person dan harus menyediakan implementasi untuk metode yang ada di interface Workable.
- **Deklarasi variabel private nim bertipe String:** Variabel nim digunakan untuk menyimpan nomor identitas mahasiswa. Bersifat private, sehingga hanya bisa diakses dari dalam kelas Student.
- **Deklarasi variabel private enrollCourse bertipe Course:** Variabel enrollCourse digunakan untuk menyimpan objek Course yang di-enroll oleh mahasiswa. Bersifat private, sehingga hanya bisa diakses dari dalam kelas Student.
- **Deklarasi konstruktor protected Student(String nim, String name, int age):** Konstruktor ini digunakan untuk membuat objek Student. Menerima tiga parameter, yaitu nim (nomor identitas mahasiswa), name (nama), dan age (usia). Kata kunci protected berarti konstruktor ini hanya bisa diakses dari subclass atau kelas lain di paket yang sama.

- **Pemanggilan konstruktor superclass dengan super(name, age):** Pernyataan `super(name, age)` digunakan untuk memanggil konstruktor dari superclass (`Person`) dan menginisialisasi atribut `name` dan `age` di kelas `Person`.
- **Inisialisasi variabel nim dengan parameter nim:** `this.nim = nim;` digunakan untuk mengisi variabel instance `nim` dengan nilai dari parameter `nim`.
- **Deklarasi metode protected getNim() bertipe String:** Metode ini mengembalikan nilai dari variabel `nim`. Kata kunci `protected` berarti metode ini bisa diakses dari subclass atau kelas lain di dalam paket yang sama.
- **Metode getNim() mengembalikan nilai nim:** `return nim;` mengembalikan nilai dari variabel `nim`.
- **Deklarasi metode protected getEnrolledCourse() bertipe Course:** Metode ini mengembalikan objek `Course` yang sedang di-enroll oleh mahasiswa. Kata kunci `protected` berarti metode ini bisa diakses oleh subclass atau kelas lain di dalam paket yang sama.
- **Metode getEnrolledCourse() mengembalikan nilai enrollCourse:** `return enrollCourse;` mengembalikan objek `Course` yang sedang di-enroll oleh mahasiswa.
- **Deklarasi metode protected enrollCourse(Course course):** Metode ini menerima parameter bertipe `Course` dan digunakan untuk mengisi variabel `enrollCourse`. Kata kunci `protected` menunjukkan metode ini bisa diakses oleh subclass atau kelas lain di dalam paket yang sama.
- **Mengisi enrollCourse dengan objek Course yang di-enroll:** `this.enrollCourse = course;` mengisi variabel `enrollCourse` dengan objek `Course` yang diterima sebagai parameter.
- **Override metode work() dari interface Workable:** Kelas `Student` harus mengimplementasikan metode `work()` dari interface `Workable`. Metode ini memberikan definisi bahwa seorang mahasiswa sedang belajar.
- **Menampilkan output "Student is studying" pada console:** Di dalam metode `work()`, pernyataan `System.out.println("Student is studying");` digunakan untuk mencetak teks ke layar.

```

1 public interface Workable {
2     public void work();
3 }
4
5

```

Penjelasan :

1. Deklarasi Interface Workable

`public interface Workable {` mendefinisikan sebuah interface bernama `Workable`. Interface adalah sebuah kontrak yang mendefinisikan metode, tetapi tidak menyediakan implementasinya. Kelas yang mengimplementasikan interface ini harus menyediakan implementasi dari metode yang dideklarasikan di dalamnya.

2. Deklarasi Metode work

`public void work();` mendeklarasikan metode `work()` yang bersifat `public` dan mengembalikan nilai `void` (tidak mengembalikan apapun). Karena ini berada dalam sebuah interface, metode ini tidak memiliki tubuh/implementasi.

3. Penutup Interface

`}` menutup deklarasi interface `Workable`.

11S2324-PBO

Laporan Praktikum

Output :

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

> cd "d:\semester 3\PBO\Praktikum\Week 4\ifs23024-pbo-praktikum-4>"

javac App.java } ; if ($?) { java App }

11S23024
Glen Rejeki Sitorus
19
Perumahan
Batam
Karisma Indah
PBO
4
Person Name: Glen Rejeki Sitorus
Person Age: 19
Student ID: 11S23024
Student Home Model: Perumahan
Student Address: Karisma Indah , Batam
Enrolled Course: PBO [4sk]
Student is studying
PS D:\SEMESTER 3\PBO\Praktikum\Week 4\ifs23024-pbo-praktikum-4> []
Java: Ready
```