BAB VII
POLYMORPHISM

Polymorphism merupakan salah satu konsep penting dalam *Object Oriented Programming* (OOP) khususnya di bahasa pemrograman Java setelah abstraction dan inheritance pada bab sebelumnya. Polymorphism sendiri memiliki arti harfiah yaitu banyak bentuk. Ada beberapa definisi berbeda tentang polymorphism yang berkaitan dengan pemrograman berorientasi objek. Sedangkan apa yang dimaksud dengan polymorphism sendiri, sebenarnya sulit untuk didefinisikan, sehingga diperlukan bab terpisah untuk menjelaskan polymorphism itu sendiri.

A. Polymorphism

Polymorphism adalah kemampuan dari suatu objek untuk merepresentasikan satu bentuk ke dalam banyak bentuk. Penggunaan paling umum dari polymorphism pada OOP terjadi ketika *reference superclass* digunakan untuk merujuk ke inharitance objek class.

Sebuah *reference variable* dapat merujuk ke objek apa pun dari jenis yang dideklarasikan atau *subtype* apa pun dari jenis yang telah dideklarasikan. *Reference variable* dapat dideklarasikan sebagai class atau *interface type.*

Contoh:

```
1  public interface vegetarian{}
2  public class Hewan{}
3  public class Rusa extends Hewan implements vegetarian{}
```

Dari contoh di atas, class Rusa dianggap polymorphic karena class memiliki multiple inharitance, sehingga pada contoh di atas berlaku:

- Rusa adalah Hewan
- Rusa adalah Vegetarian
- Rusa adalah Rusa
- Rusa adalah Objek

Ketika diterapkan fakta *reference variable* di atas ke objek *reference Rusa,* maka dideklarasikan sebagai berikut:

```
1   Rusa rusa =  new Rusa();
2   Hewan hewan = rusa;
3   Vegetarian vegan = rusa;
4   Object obj = rusa;
```

Bisa kita lihat semua *reference variable* seperti hewan, rusa, vegan, dan obj merujuk pada objek Rusa yang sama.

B. Jenis-jenis Polymorphism dalam Pemrograman

Polymorphism dalam *Object-Oriented Programming* (OOP) dibagi menjadi dua jenis utama:

1. Runtime Polymorphism (Dynamic Polymorphism)

Juga dikenal sebagai **Method Overriding**. Pada jenis ini, keputusan pemanggilan metode dibuat saat runtime, biasanya menggunakan konsep inheritance.

Contoh Method Override:

```
1  class Hewan {
2      void bersuara() {
3          System.out.println("Hewan mengeluarkan suara");
4      }
5  }
6
7  class Kucing extends Hewan {
8      @Override
9      void bersuara() {
10         System.out.println("Meong Meong");
11     }
12 }
13
14 public class Main {
15     public static void main(String[] args) {
16         Hewan hewan1 = new Kucing();  // Polymorphism
17         hewan1.bersuara(); // Output: Meong Meong
18     }
19 }
20
```

Pada contoh ini, meskipun variabel *hewan1* bertipe *Hewan*, ia menjalankan metode *bersuara()* dari *Kucing*.

2. Compile-Time Polymorphism (Static Polymorphism)

Juga dikenal sebagai **Method Overloading** dan **Operator Overloading**. Pada jenis ini, keputusan pemanggilan metode dibuat saat **kompilasi**.
Contoh Method Overloading:

```java
class Perkalian {
    // Metode dengan dua parameter
    int multiply(int a, int b) {
        return a * b;
    }

    // Metode dengan tiga parameter (Overloading)
    int multiply(int a, int b, int c) {
        return a * b * c;
    }
}

public class Main {
    public static void main(String[] args) {
        Perkalian p = new Perkalian();
        System.out.println(p.multiply(2, 3));      // Output: 6
        System.out.println(p.multiply(2, 3, 4));   // Output: 24
    }
}
```

Pada contoh di atas, metode multiply dipanggil berdasarkan jumlah argumen yang diberikan.

C. Virtual Method

Pada bagian ini, akan ditunjukkan bagaimana behaviour dari overridden method pada Java memungkinkan untuk dimanfaatkan pada polymorphism saat mendesain class.

Pada bab sebelumnya juga telah dijelaskan mengenai overriding, dimana subclass dapat override sebuah method dari superclass-nya. Overridden method pada dasarnya tak terlihat pada superclass, dan tidak terpanggil kecuali subclass-nya menggunakan kata kunci super dalam overriding method.
Contoh:

```java
// Superclass
public class Manusia {
    void makan() {
        System.out.println("Manusia makan");
    }

    void tidur() {
        System.out.println("Manusia tidur");
    }

    void bergerak() {
        System.out.println("Manusia bergerak");
    }
}
```

```java
public class Mahasiswa extends Manusia{
    @Override
    void makan(){
        System.out.println("Mahasiswa makan");
    }

    @Override
    void tidur(){
        System.out.println("Mahasiswa tidur");
    }

    @Override
    void bergerak(){
        System.out.println("Mahasiswa bergerak");
    }
}
```

```java
public class Asistensi extends Manusia{
    @Override
    void makan(){
        System.out.println("Asistensi makan");
    }
    @Override
    void tidur(){
        System.out.println("Asistensi tidur");
    }

    @Override
     void bergerak(){
        System.out.println("Asistensi bergerak");
}
```

```java
public class Programmer extends Manusia {
    @Override
    public void makan() {
        System.out.println("Programmer makan");
    }

    @Override
    public void tidur() {
        System.out.println("Programmer tidur");
    }

    @Override
    void bergerak() {
        System.out.println("Programmer bergerak");
    }
}
```

```java
public class Test {
    public static void main(String[] args) {
        Manusia [] manusia = new Manusia[4];
        manusia[0] = new Mahasiswa();
        manusia[1] = new Asistensi();
        manusia[2] = new Programmer();

        for (int i = 0; i < manusia.length; i++) {
            manusia[i].makan();
            manusia[i].tidur();
            manusia[i].bergerak();
            System.out.println();
        }
    }
}
```