VERSION 2.2

FEBRUARI, 2023

PEMROGRAMAN BERORIENTASI OBJEK

MODUL 5 - POLYMORPHISM

DISUSUN OLEH:

Muhammad Nizar Zulmi Rohmatulloh

Jody Yuantoro

DIAUDIT OLEH:

Aminudin, S.Kom., M.Cs.

PRESENTED BY: TIM LAB. IT

UNIVERSITAS MUHAMMADIYAH MALANG

PEMROGRAMAN BERORIENTASI OBJEK

TUJUAN

1. Mahasiswa dapat memahami konsep polimorfisme.
2. Mahasiswa dapat memahami *instance* secara mendalam.

TARGET MODUL

1. Mahasiswa dapat membuat dan mengimplementasikan polimorfisme.

PERSIAPAN

1. Java Development Kit.
2. Text Editor / IDE (Visual Studio Code, Netbeans, Intellij IDEA, atau yang lainnya).

KEYWORDS

- Polymorphism
- Type & Object Casting
- Polymorphic Arguments
- Collection

TEORI

- **Polymorphism**

  Polymorphism artinya banyak bentuk. Maksudnya apa? Kita ketahui bahwa semua *class* dalam Java (secara implisit) meng-*extends* class Object, jadi dalam Java semua class adalah object. Seperti contoh sebelumnya, Kucing adalah seekor hewan, kucing juga adalah sebuah objek. Contoh lainnya adalah Anda sendiri, Anda bisa dibilang seorang mahasiswa saat di lingkungan kampus, atau juga bisa disebut seorang warga negara Indonesia, atau juga seorang anak dari orang tua Anda. Itulah yang dimaksud banyak bentuk / polimorfisme dalam OOP.

- **Object Casting**

  Sebelumnya di pemrograman dasar, kalian seharusnya sudah mempelajari *type casting*, di dalamnya juga terdapat *narrowing* dan *widening cast/conversion*, kalau belum silakan di-*review*/dipelajari sendiri.

  Object Casting digunakan untuk mengubah tipe dari suatu objek. Seperti yang telah kita pelajari sebelumnya kita dapat membuat tipe data sendiri dari kumpulan *Primitive Data Type* (`int`, `char`, `float`, dll) yang disebut *Reference Data Type*. Hewan, Kucing, dan Anjing sebelumnya dapat kita gunakan sebagai tipe data referensi. *Object casting* disini berperan untuk mengubah tipe data dari subclass ke superclass (*upcasting*) atau sebaliknya (*downcasting*). Contohnya:

  - Upcasting

```
Kucing k = new Kucing();
Hewan h = k; // di sini terjadi upcasting
```

  - Downcasting

```
Kucing k = new Kucing();
Hewan h = k; // implicit casting
Kucing kucing = (Kucing) h; // downcasting / explicit casting
```

Downcasting juga disebut *explicit casting*, karena objek tujuan dari casting harus ditulis secara eksplisit yaitu dalam tanda kurung sebelum objek yang akan di-casting.

Di dalam OOP pada Java terdapat 2 jenis polimorfisme, yaitu statis dan dinamis, yang keduanya sudah dipelajari di modul sebelumnya.

- **Static Polymorphism**

  Static Polymorphism (Polimorfisme statis) pada Java adalah saat terjadi *method overloading* yang telah kita pelajari di modul sebelumnya.

- **Dynamic Polymorphism (virtual method invocation)**

  Dynamic polymorphism (polimorfisme dinamis) dapat kita lakukan dengan cara menerapkan *method overriding* yang telah kita pelajari di modul sebelumnya. Ini terjadi saat ada pembuatan objek dengan tipe superclass (*parent class*) tetapi memanggil konstruktor dari subclass (*child class)* atau juga pemanggilan method-nya.

  Contoh:

  Kita memiliki kelas Vehicle.

```
package edu.praktikum.pbo.modul5;

public class Vehicle {

    public void move() {
        System.out.println("Vehicle is moving");
    }

}
```

Dan kelas Car yang merupakan turunan dari kelas Vehicle.

```
package edu.praktikum.pbo.modul5;

public class Car extends Vehicle {

    @Override
    public void move() {
        System.out.println("Car is accelerating and moving");
    }

}
```

Dan saat kita jalankan:

```
package edu.praktikum.pbo.modul5;

public class Main {
    public static void main(String[] args) {
        Vehicle car = new Car();
        car.move(); // Output: Car is accelerating and moving

        car = new Vehicle();
        car.move(); // Output: Vehicle is moving
    }
}
```

- Dari percobaan di atas dapat kita lihat bahwa, pertama kita membuat objek **`car`** yang bertipe **`Vehicle`**, namun memegang referensi ke class **`Car`**.
- Pada saat pemanggilan method **`move()`** yang pertama, yang dieksekusi adalah method **`move()`** milik class **`Car`**.
- Kemudian objek **`car`** dikembalikan referensinya ke class **`Vehicle`**.
- Setelah itu pemanggilan method **`move()`** yang dieksekusi adalah milik **`Vehicle`**.

Ini dinamakan *Virtual Method Invocation*, disebut virtual karena antara method yang dikenali oleh *compiler* dan method yang dijalankan oleh JVM berbeda. Saat *compile time*, *compiler* akan mengenali method **`move()`** yang akan dipanggil adalah method **`move()`** yang ada di class **`Vehicle`**, karena objek **`car`** bertipe **`Vehicle`**. Tetapi saat dijalankan (*runtime*), maka yang dijalankan oleh JVM justru method **`move()`** yang ada di class **`Car`**. Tipe data sebelah kiri / sebelum variabel dapat juga berupa interface.

Secara sintaks:

```
Interface namaObjek = new ClassYangImplements();
```

Contoh:

```
Flyer f = new Superman();
Flyer f = new Pesawat();
```

- **Polymorphic Arguments**

  Kita ketahui bahwa tipe data di Java bukan hanya tipe data primitif, kita juga bisa membuat tipe referensi (class). *Method* juga dapat memiliki parameter bertipe referensi umumnya merupakan tipe class yang menerapkan *inheritance*. Contoh akan kita tambahkan kelas baru pada kode di submateri *dynamic polymorphism*:

```
package edu.praktikum.pbo.modul5;

public class Truck extends Vehicle {

    @Override
    public void move() {
        System.out.println("Truck is accelerating and carrying
loads");
    }

}
```

```
package edu.praktikum.pbo.modul5;
```

```
public class Mechanic {

    public void repair(Vehicle vehicle) {
        if (vehicle instanceof Car) {
            System.out.println("Repairing car");
        } else if (vehicle instanceof Truck) {
            System.out.println("Repairing truck");
        } else {
            System.out.println("Repairing vehicle");
        }
    }

}
```

Lalu pada kelas Main kita ganti menjadi:

```
package edu.praktikum.pbo.modul5;

public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        Truck truck = new Truck();
        Mechanic mechanic = new Mechanic();

        mechanic.repair(car); // Output: Repairing car
        mechanic.repair(truck); // Output: Repairing truck
    }
}
```

Method **repair(Vehicle vehicle)** di atas dapat menerima objek **car** dan **truck** karena keduanya merupakan keturunan Vehicle. Kemudian ia akan menjalankan sesuai apa yang dicek dengan kondisi `instanceof`.

- **Heterogeneous Collection**

  Di pemrograman dasar Anda sudah mengenal apa itu array. Dengan adanya konsep polimorfisme, maka variabel array bisa dibuat heterogen. Artinya di dalam array tersebut bisa berisi berbagai macam objek yang berbeda tetapi dengan memperhatikan relasi is-a. Contoh, ubah Main di submateri Polymorphic Arguments menjadi seperti berikut:

```
package edu.praktikum.pbo.modul5;

public class Main {
    public static void main(String[] args) {
        Vehicle[] vehicles = new Vehicle[3];
        vehicles[0] = new Vehicle();
        vehicles[1] = new Car();
        vehicles[2] = new Truck();

        for (Vehicle vehicle : vehicles) {
            vehicle.move();
        }
    }
}

Output:
Vehicle is moving
Car is accelerating and moving
Truck is accelerating and carrying loads
```

Pada kode di atas, kita mengelompokkan beberapa data ke satu variable yang sama. Data ke-1 berisi objek **`Vehicle`** data ke-2 berisi objek **`Car`** sedangkan data ke-3 berisi objek **`Truck`** yang keduanya merupakan turunan dari kelas **`Vehicle`**. Kemudian kita memanggil *method* **`move()`** dengan cara ***foreach***, hal ini tidak menimbulkan *error* karena semua objek yang tersimpan di array vehicles memiliki *method* **`move()`**.

Jika menggunakan perulangan **`for`** biasa:

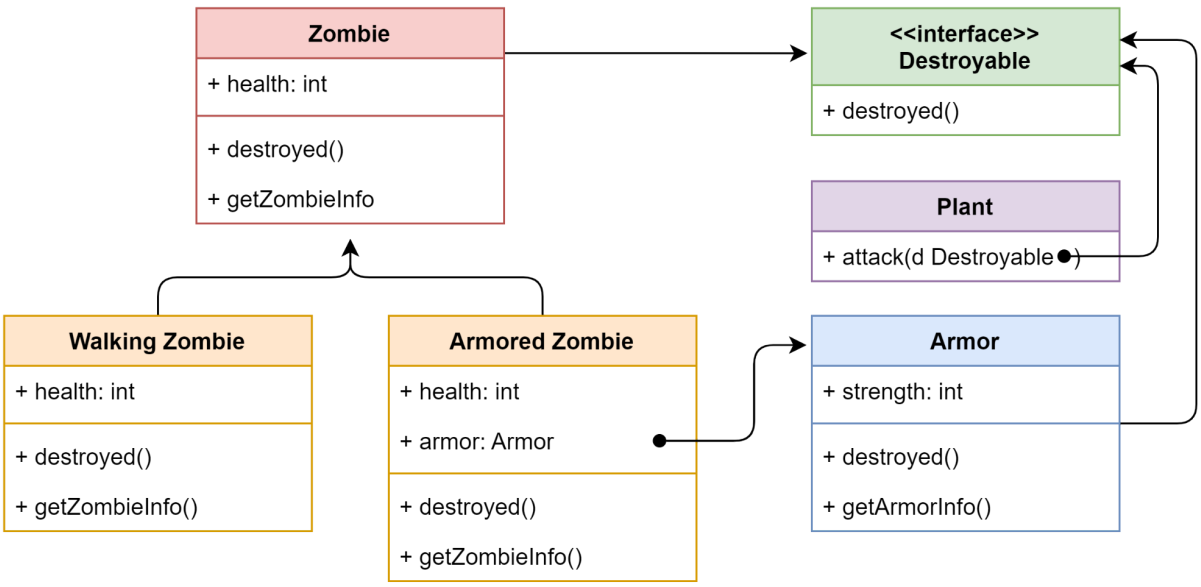
```
for (int i = 0; i < vehicles.length; i++) {
    vehicles[i].move();
}
```

CODELAB

Dalam suatu permainan, Plant dapat melindungi diri dengan cara menyerang Zombie. Di permainan ini terdapat dua jenis Zombie, Walking Zombie dan Armored Zombie. Kedua Zombie tersebut saat diserang memiliki efek penyerangan yang berbeda:

- Pada Walking Zombie
  - Setiap diserang health akan berkurang 40%.
- Pada Armored Zombie
  - Health tidak akan berkurang sebelum armor strength mencapai 0.
  - Setiap penyerangan armor strenght akan berkurang 20%.
  - Setelah armor strength mencapai 0, setiap penyerangan health akan berkurang 20%.

Kira-kira jika digambarkan dalam UML adalah seperti ini:

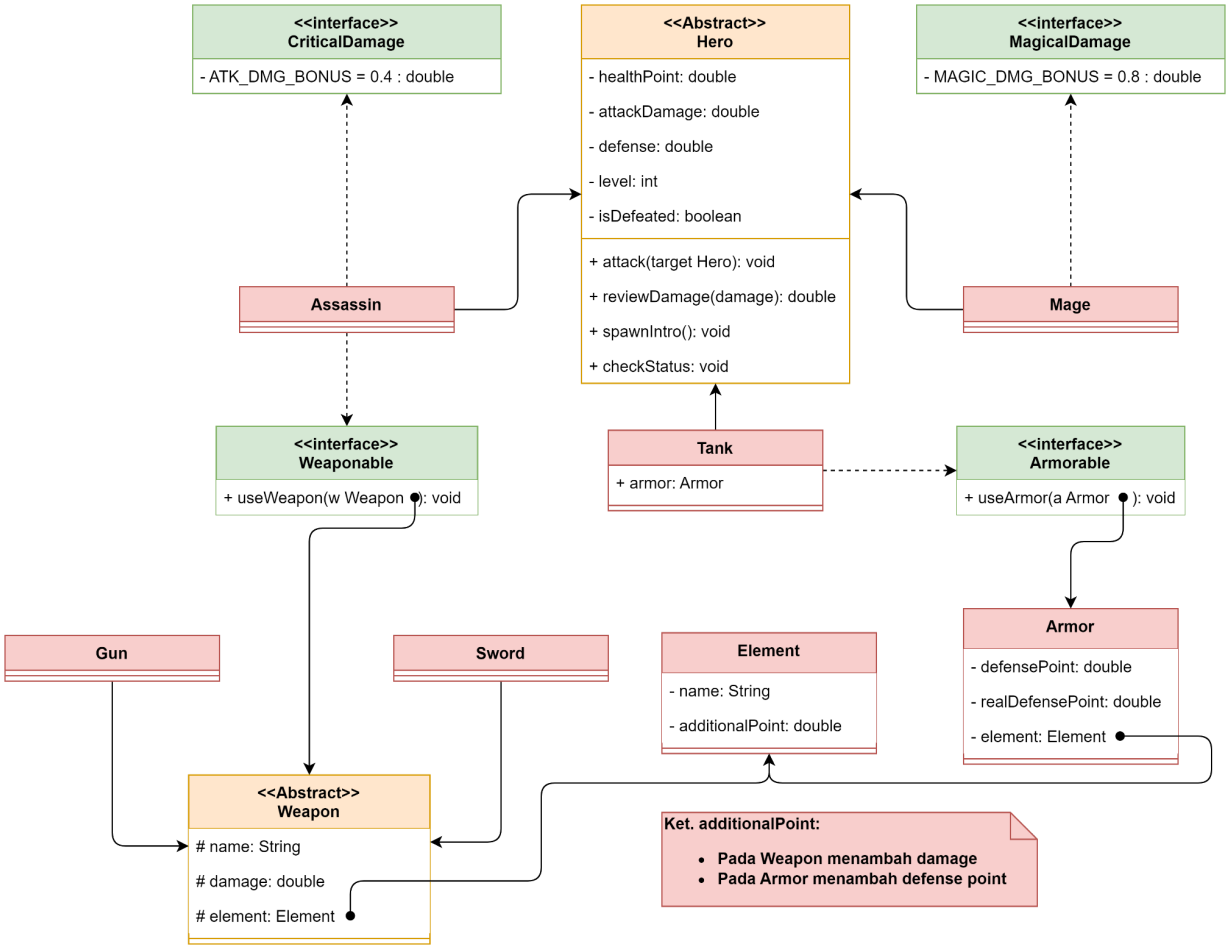


Hint: Objek plant akan memanggil method `attack(d Destroyable)` yang berisi objek yang meng-*implements* `Destroyable.` Method ini akan memanggil method `destroyed()` pada suatu objek yang meng-*implements* `Destroyable`. Setiap penyerangan akan menampilkan status health Zombie beserta strength Armor-nya.

Contoh output (tampilkan juga WalkingZombie):

```
====== Zombie sebelum diserang ======
Zombie: ArmoredZombie
Health: 100
Armor: 100

====== Plant menyerang ArmoredZombie ======
====== Zombie Info ======
Zombie: ArmoredZombie
Health: 100
Armor: 80

====== Plant menyerang ArmoredZombie ======
====== Zombie Info ======
Zombie: ArmoredZombie
Health: 100
Armor: 64

Skip
Skip

====== Plant menyerang ArmoredZombie ======
====== Zombie Info ======
Zombie: ArmoredZombie
Health: 2
Armor: 0

====== Plant menyerang ArmoredZombie ======
====== Zombie Info ======
Zombie: ArmoredZombie
Health: 1
Armor: 0

====== Plant menyerang ArmoredZombie ======
====== Zombie Info ======
Zombie: ArmoredZombie
Health: 0
Armor: 0
====== Zombie Mati ======
```

TUGAS



Buatlah sebuah fighting game sederhana sesuai dengan class diagram diatas dengan syarat dan ketentuan:

A. Konsep:

- Character memiliki healthPoint, attackDamage, defense, level, dan lifeStatus
  - healthPoint: status yang menunjukkan jumlah satuan Kesehatan karakter game
  - attackDamage: nilai serangan yang akan diberikan untuk mengurangi healthpoint lawan
  - defense: ketahanan hero, digunakan ketika mendapat serangan dari lawan.
  - level: tingkat kemahiran/ kehebatan hero
  - isDefeated: status jika terkalahkan.

- Method attack(), reviewDamage(), dan checkStatus() merupakan method concrete, sedangkan spawnIntro() merupakan abstract method.
- Method attack(target : Hero) : menyerang target
- Method reviewDamage(damage : double) : mengurangi healthPoint sebesar damage yang telah dikurangi defense atau dengan kata lain bisa disebut realDamage
    - Rumus untuk realDamage : realDamage = attackDamage – defense
    - healthPoint tidak boleh kurang dari 0 dan jika healthpoint menyentuh nilai 0 maka lifeStatus akan berubah menjadi false atau Hero tersebut telah mati dan pertandingan selesai
- Method spawnIntro() : dialog awal Hero ketika telah memasuki giliran Hero tersebut.
    - Contoh : Hero Tank : System.out.println(“Kill me if u can !!”);
- Method checkStatus() : menghasilkan data-data atribut Hero seperti sisa healthPoint
- Spesifikasi dasar Hero adalah isDefeated: false dan penyesuaian tiap jenis Hero sebagai berikut:
    - Assassin:
        - healthPoint : 3000
        - defense: 300
        - attackDamage: 800
    - Tank :
        - healthPoint : 7000
        - defense: 500
        - attackDamage: 500
    - Mage :
        - healthPoint : 2500
        - defense: 200
        - attackDamage: 700

- Tiap jenis Hero memiliki constructor dengan 1 parameter (level : int) yang digunakan untuk menginialisasi nilai-nilai atribut.
- level pada parameter constructor digunakan untuk menambahkan nilai atribut Hero jika level > 0, dengan nilai tiap 1 levelnya:
    - Assassin:
        - (+90 healthPoint)
        - (+15 defense)
        - (+30 attackDamage)
    - Tank :
        - (+200 healthPoint)
        - (+15 defense)
        - (+20 attackDamage)
    - Mage :
        - (+85 healthPoint)
        - (+10 defense)
        - (+35 attackDamage)
- Untuk Hero Asassin, mendapatkan Critical Damage atau tambahan attackPoint selain dari level. Pada Constructor menambahkan attackDamage sebesar attackDamage * bonusDamage(0.4) pada interface CriticalDamage, dapat juga dituliskan seperti :
    - attackDamage += attackDamage * bonusDamage
- Hero Assassin dapat memiliki senjata (*Weapon*) senjata tersebut digunakan dengan method useWeapon() efeknya adalah menambah damage. Weapon tersebut juga dapat memiliki element (tetapi juga harus bisa tanpa memasang Element) berupa api, tanah, air, dan angin. Penggunaan Element menambah attackDamage pada hero assassin.
- Hero Tank dapat memiliki pelindung (Armor) yang memiliki defensePoint / strength, apabila masih ada strength pada armor maka HP hero tank tidak berkurang, dan hanya akan berkurang ketika diserang dan armor strength-nya 0.

- Untuk Hero Mage, mendapatkan Magical Damage atau tambahan attackPoint selain dari level. Pada Constructor menambahkan attackDamage sebesar attackDamage * magicDamageBonus(0.8) pada interface MagicalDamage, dapat juga dituliskan seperti :
  - attackDamage += attackDamage * magicalDamageBonus

B. Teknis

- Boleh melakukan improvisasi dengan tetap memenuhi ketentuan di atas atau menjadikannya lebih baik (*more advanced*) akan menambah nilai. Seperti:
  - Tipe Element (api, tanah, air, angin) dijadikan concrete class sendiri dengan cara class Element dijadikan *abstract* dan di-*extends* oleh FireElement, WaterElement dan seterusnya alih-alih pengecekan string langsung / atribut name.
  - Menambahkan getter dan setter jika perlu (menerapkan prinsip enkapsulasi) ataupun method lain yang membuat kode anda jadi lebih mudah dibaca.
  - HP, defense, atk di-*random generate* dengan ketentuan anda sendiri, misal HP kelipatan 100 secara berdekatan (Hero 1 random HP 1500, Hero 2 random HP 1400)
- Akan lebih baik kalau file-file ditempatkan sesuai dengan *package*-nya.
- Implementasi objek game yang telah dibuat berada di ***class Main***. Game tersebut nantinya akan berlangsung dengan mode 1v1 dengan giliran attack bergantian dan akan berhenti ketika salah satu Hero mati. Hero yang masih hidup adalah pemenang. Contoh output dapat dilihat di sini (tidak harus sama persis).

RUBRIK PENILAIAN

ASPEK PENILAIAN	POIN
Codelab	15
Tugas	40
Pemahaman	45
TOTAL	**100**

Selamat Mengerjakan
Tetap Semangat