Polymorphism

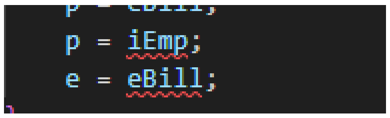
Name	Davis Maulana Hermanto
Class	TI 2i
NIM	2241720255
Subject	Object Oriented Programming
Type	Assignment
Semester	Semester 3
Time	@November 15, 2023

Experiment 1

1. What classes are derivatives of the Employee class?
    a. There are 2 classes, named PermanentEmployee class and InternshipEmployee class.
2. What classes implement the Payable interface?
    a. There are 2 classes, named PermanentEmployee class and ElectricityBill class.
3. Pay attention to the Tester1 class, lines 10 and 11. Why e, can be filled in with the pEmp object (an object from the PermanentEmployee class) and the iEmp object (which is an object from the class
InternshipEmploye) ?
    a. Because pEmp(PermanentEmployee) and iEmp(InternshipEmployee) are derivatives of the Employee class.
4. Pay attention to the Tester1 class, lines 12 and 13. Why p, can be filled with object pEmp (is an object from class PermanentEmployee) and the eBill object (is an object of the class Electricity bills) ?

   a. Because pEmp and eBill implement to Payable interface.

5. Try adding the syntax:
   p = iEmp;
   e = eBill;
   on lines 14 and 15 (the last lines in the main method)! What cause the error?
   a. Because there is no relationship between iEmp to p and eBill to e (iEmp doesn't implement to p and eBill is not derivative of the Employee class (e)).



```
p = iEmp;
e = eBill;
```

6. Draw conclusions about the basic concepts/forms of polymorphism!
   - Polymorphism is the concept that an object can have many forms, For example, in the Tester1 class, pEmp objects and iEmp objects can be filled into e variables of type Employee and Payable, pEmp objects and eBill objects can be filled into p variables of type Payable, and eBill objects cannot be filled into e variables of type Employee and iEmp objects cannot be filled into p variables of type Payable.

   1. **Flexibility and Modularity**
      - Polymorphism provides flexibility in the use of objects, allowing objects to have multiple forms or behaviors.
   2. **Inheritance and Substitution:**
      - Polymorphism is often related to Inheritance, where a subclass can replace or extend the behavior of its parent class.
   3. **Interfaces and implementation:**
      - The interface allows polymorphism without relying on class hierarchies, allowing objects from different classes but implementing the same interface to be used uniformly.

Experiment 2

1. Take a look at the Tester2 class above, why the calls to e.getEmployeeInfo() on line 8 and pEmp.getEmployeeInfo() on line 10 produce the same result?
   a. Because they both refer to the same object which is an instance of PermanentEmployee, and dynamic binding ensures that the appropriate methods of the actual class are called at runtime.
2. Why is the e.getEmployeeInfo() method invocation referred to as a virtual method invocation, whereas pEmp.getEmployeeInfo() is not?
   a. The method call e.getEmployeeInfo() is called a virtual method call because the actual type of object that will run the method implementation is determined dynamically at runtime. At compile time, the type of variable e is declared as type Employee, but the actual object associated with the variable is an object of a derived class, PermanentEmployee.
3. So what is virtual method invocation? Why is is it called virtual?
   a. Virtual method invocation is an overriding method invocation of a polymorphism object. It is called virtual because the method recognized by the compiler and the method executed by the JVM are different.

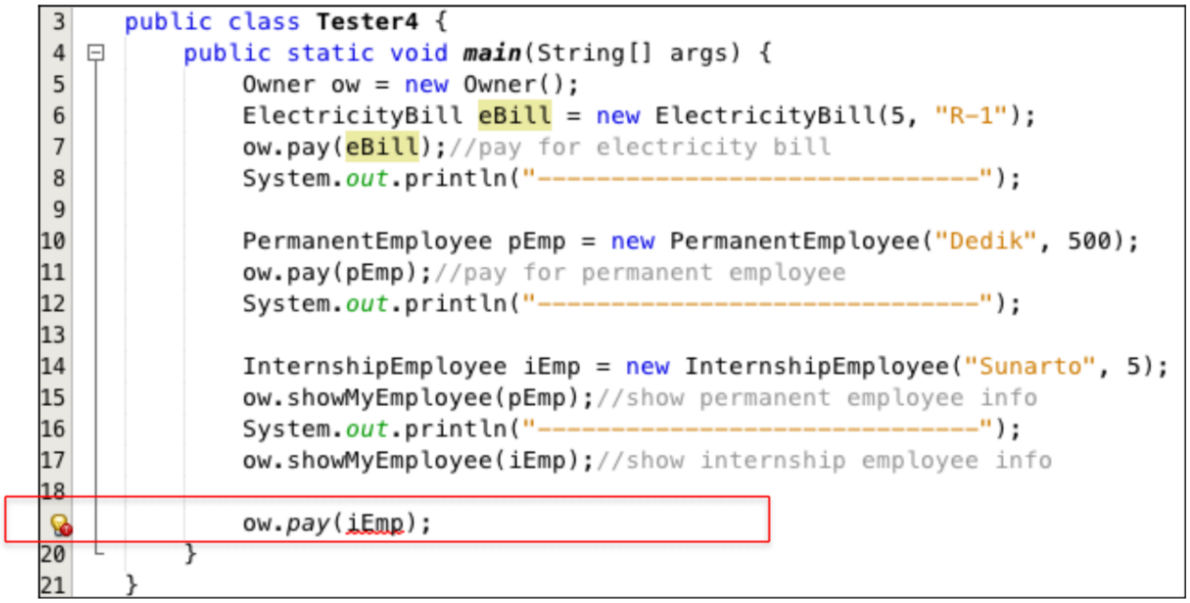
Experiment 3

1. Notice the array e on line 8, why it can be filled with objects of different types, namely the pEmp object (object of PermanentEmployee) and iEmp object (object of InternshipEmployee) ?
   a. Because array e is an array of type Employee. Since PermanentEmployee and InternshipEmployee are subclasses of Employee, objects of these two classes can be considered as objects of the more general type Employee.
2. Notice also line 9, why the p array is also populated with objects objects of different types, namely the pEmp object (object of PermanentEmployee) and eBill object (object from ElectricityBilling)?
   a. Because array p is an array of type Payable, and both PermanentEmployee and ElectricityBill are implemented from interface Payable.
3. Look at line 10, why is there an error?

a. Because eBill (ElectricityBill class) doesn't extend to Employe class (parent class).

Experiment 4

1. Consider the Tester4 class line 7 and line 11, why can the ow.pay(eBill) and ow.pay(pEmp) calls be made, when if you pay attention to the pay() method in the Owner class has an argument/parameter of type Payable? If you look in more detail eBill is an object of the ElectricityBill and pEmp is an object of PermanentEmployee?
   a. Because both eBill and pEmp objects implement the Payable interface.
2. So what is the purpose of creating an argument of type Payable in the pay() method inside the Owner class?
   a. The purpose of creating an argument of type Payable in the pay() method inside the Owner class is to utilize the concept of polymorphism and allow the method to accept objects from various classes that implement the Payable interface.
3. Try on the last line of the main() method in the Tester4 class to add the command ow.pay(iEmp);



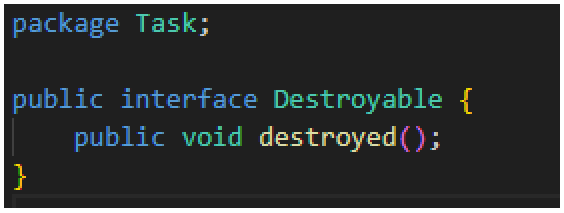
```
 3  public class Tester4 {
 4      public static void main(String[] args) {
 5          Owner ow = new Owner();
 6          ElectricityBill eBill = new ElectricityBill(5, "R-1");
 7          ow.pay(eBill);//pay for electricity bill
 8          System.out.println("------------------------------");
 9
10          PermanentEmployee pEmp = new PermanentEmployee("Dedik", 500);
11          ow.pay(pEmp);//pay for permanent employee
12          System.out.println("------------------------------");
13
14          InternshipEmployee iEmp = new InternshipEmployee("Sunarto", 5);
15          ow.showMyEmployee(pEmp);//show permanent employee info
16          System.out.println("------------------------------");
17          ow.showMyEmployee(iEmp);//show internship employee info
18
            ow.pay(iEmp);
20      }
21  }
```

Why is there an error?

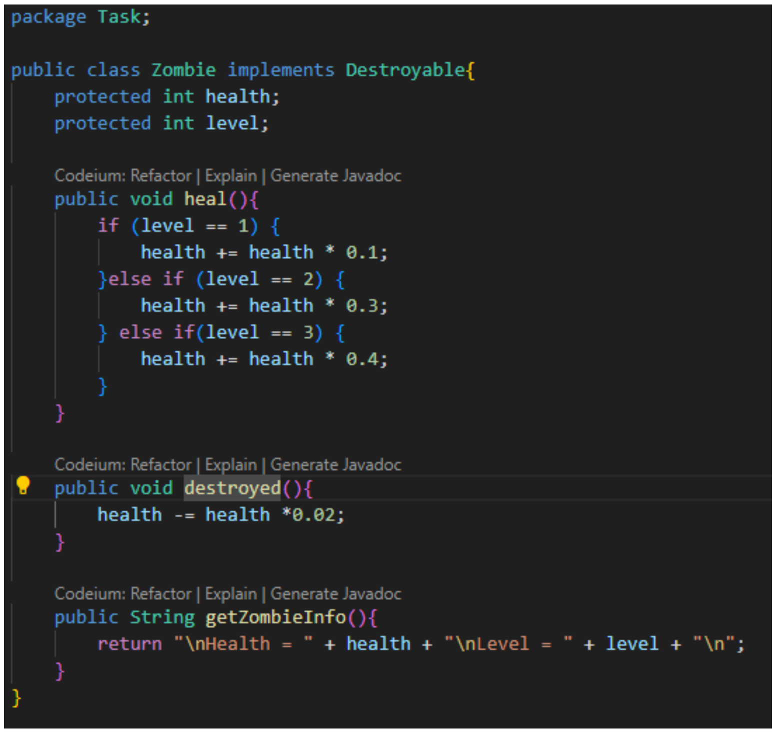
- Because the pay() method in the Owner class only handles objects that implement the Payable interface, and InternshipEmployee does not directly implement the interface.

4. Note the Owner class, required for whether the syntax p instanceof ElectricityBill on line 6?
    a. is used to perform a check on the type of object p whether it is an instance of the ElectricityBill class or not. This check utilizes the instanceof operator, which returns true if the checked object is an instance of the specified class, and false otherwise.
5. Look back at the Owner class line 7, what is the casting of the object there (ElectricityBill eb = (ElectricityBill) p) required for? Why should the object p of type Payable be cast into the object of type ElectricityBill?
    a. Casting the object on line 7 (ElectricityBill eb = (ElectricityBill) p) is necessary because the parameter p received by the pay method has the Payable data type, and we want to access methods or attributes that exist only in the ElectricityBill class.

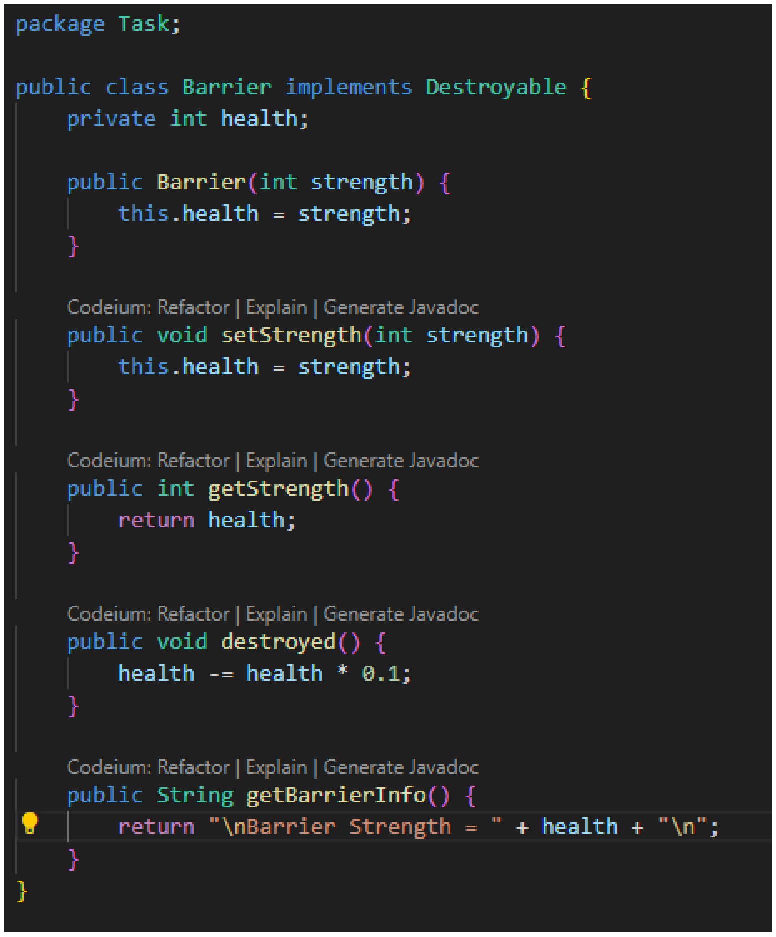
Task



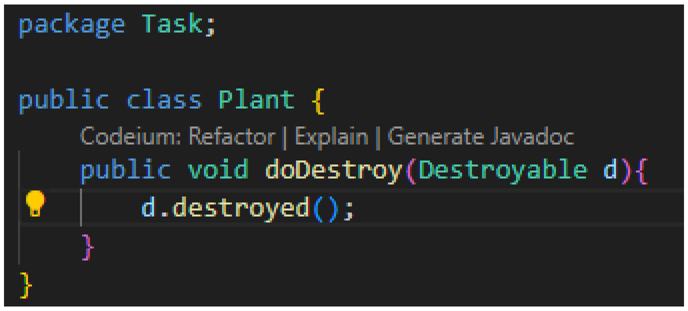
```
package Task;

public interface Destroyable {
    public void destroyed();
}
```



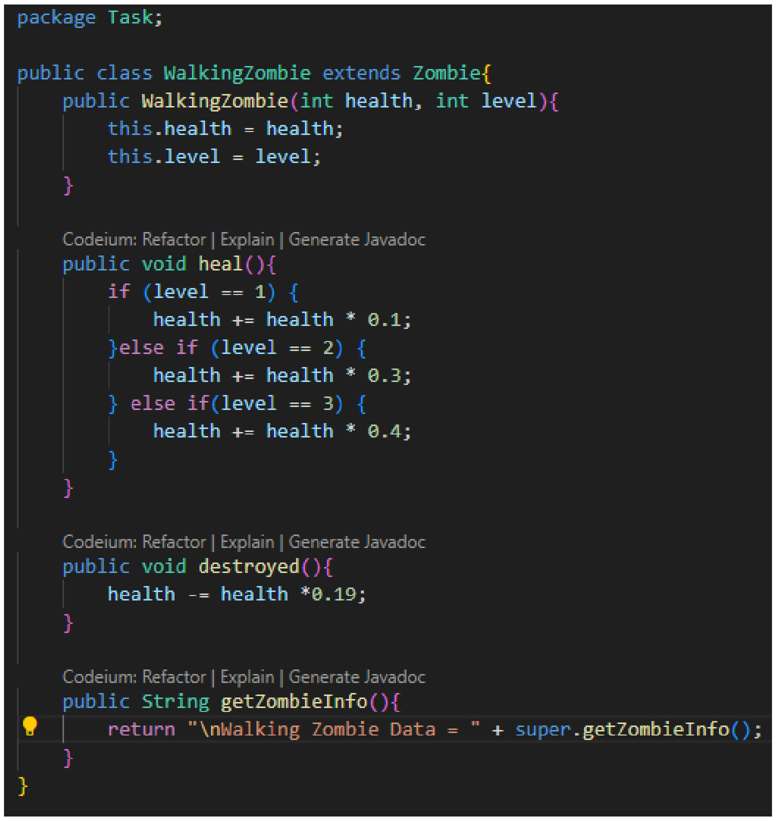
```
package Task;

public class Zombie implements Destroyable{
    protected int health;
    protected int level;

    Codeium: Refactor | Explain | Generate Javadoc
    public void heal(){
        if (level == 1) {
            health += health * 0.1;
        }else if (level == 2) {
            health += health * 0.3;
        } else if(level == 3) {
            health += health * 0.4;
        }
    }

    Codeium: Refactor | Explain | Generate Javadoc
    public void destroyed(){
        health -= health *0.02;
    }

    Codeium: Refactor | Explain | Generate Javadoc
    public String getZombieInfo(){
        return "\nHealth = " + health + "\nLevel = " + level + "\n";
    }
}
```



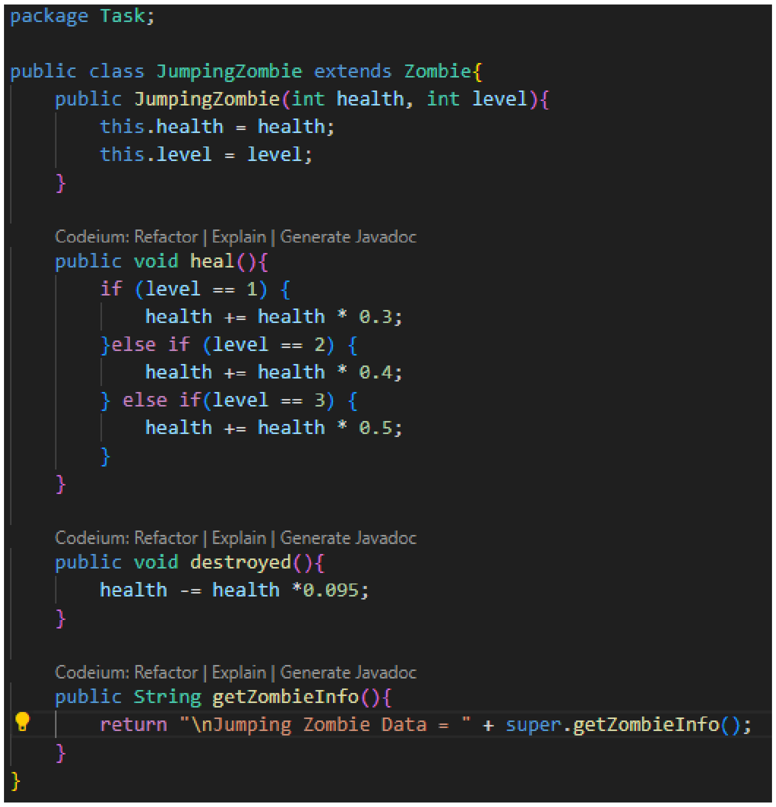
```
package Task;

public class Barrier implements Destroyable {
    private int health;

    public Barrier(int strength) {
        this.health = strength;
    }

    Codeium: Refactor | Explain | Generate Javadoc
    public void setStrength(int strength) {
        this.health = strength;
    }

    Codeium: Refactor | Explain | Generate Javadoc
    public int getStrength() {
        return health;
    }

    Codeium: Refactor | Explain | Generate Javadoc
    public void destroyed() {
        health -= health * 0.1;
    }

    Codeium: Refactor | Explain | Generate Javadoc
    public String getBarrierInfo() {
        return "\nBarrier Strength = " + health + "\n";
    }
}
```



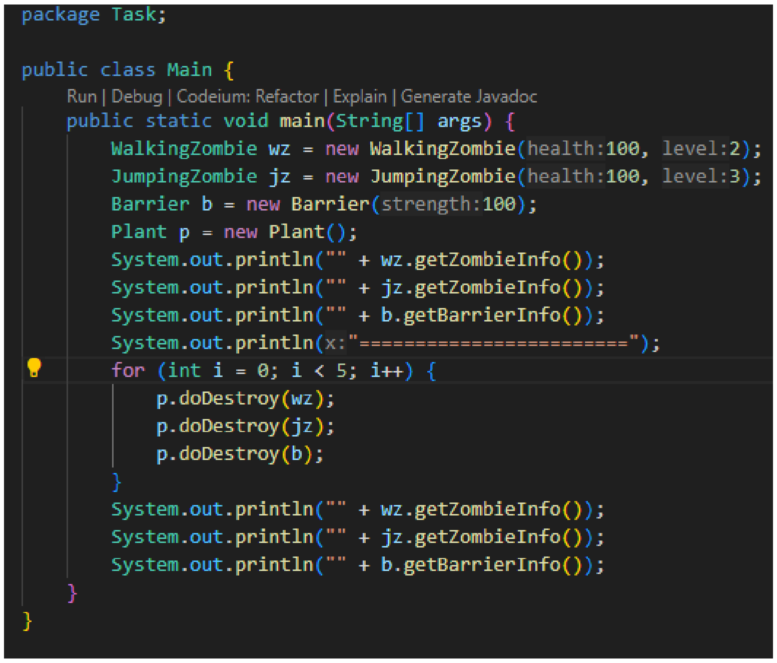
```
package Task;

public class Plant {
    Codeium: Refactor | Explain | Generate Javadoc
    public void doDestroy(Destroyable d){
        d.destroyed();
    }
}
```



```
package Task;

public class WalkingZombie extends Zombie{
    public WalkingZombie(int health, int level){
        this.health = health;
        this.level = level;
    }

    Codeium: Refactor | Explain | Generate Javadoc
    public void heal(){
        if (level == 1) {
            health += health * 0.1;
        }else if (level == 2) {
            health += health * 0.3;
        } else if(level == 3) {
            health += health * 0.4;
        }
    }

    Codeium: Refactor | Explain | Generate Javadoc
    public void destroyed(){
        health -= health *0.19;
    }

    Codeium: Refactor | Explain | Generate Javadoc
    public String getZombieInfo(){
        return "\nWalking Zombie Data = " + super.getZombieInfo();
    }
}
```



```
package Task;

public class JumpingZombie extends Zombie{
    public JumpingZombie(int health, int level){
        this.health = health;
        this.level = level;
    }

    Codeium: Refactor | Explain | Generate Javadoc
    public void heal(){
        if (level == 1) {
            health += health * 0.3;
        }else if (level == 2) {
            health += health * 0.4;
        } else if(level == 3) {
            health += health * 0.5;
        }
    }

    Codeium: Refactor | Explain | Generate Javadoc
    public void destroyed(){
        health -= health *0.095;
    }

    Codeium: Refactor | Explain | Generate Javadoc
    public String getZombieInfo(){
        return "\nJumping Zombie Data = " + super.getZombieInfo();
    }
}
```



```
package Task;

public class Main {
    Run | Debug | Codeium: Refactor | Explain | Generate Javadoc
    public static void main(String[] args) {
        WalkingZombie wz = new WalkingZombie(health:100, level:2);
        JumpingZombie jz = new JumpingZombie(health:100, level:3);
        Barrier b = new Barrier(strength:100);
        Plant p = new Plant();
        System.out.println("" + wz.getZombieInfo());
        System.out.println("" + jz.getZombieInfo());
        System.out.println("" + b.getBarrierInfo());
        System.out.println(x:"========================");
        for (int i = 0; i < 5; i++) {
            p.doDestroy(wz);
            p.doDestroy(jz);
            p.doDestroy(b);
        }
        System.out.println("" + wz.getZombieInfo());
        System.out.println("" + jz.getZombieInfo());
        System.out.println("" + b.getBarrierInfo());
    }
}
```