**JOBSHEET**

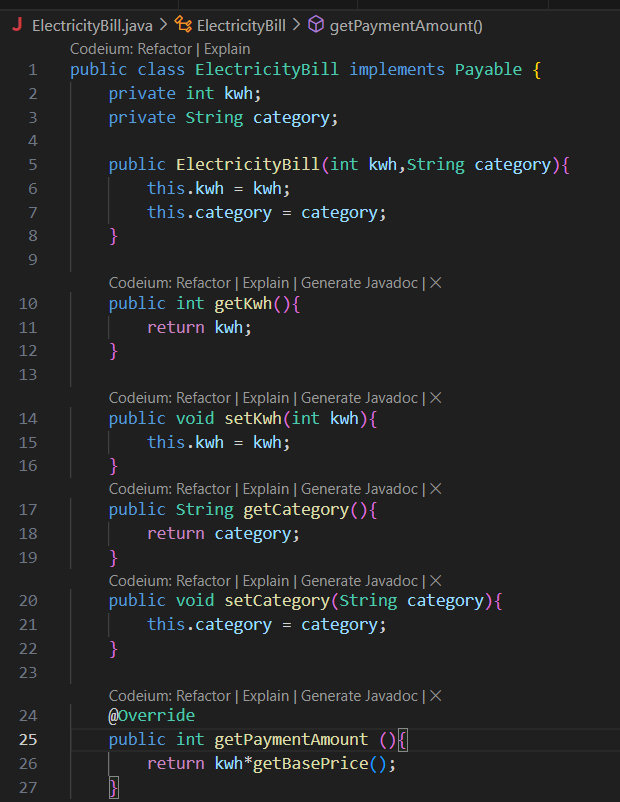
**POLYMORPHISM**

Erwan Majid 08 2i

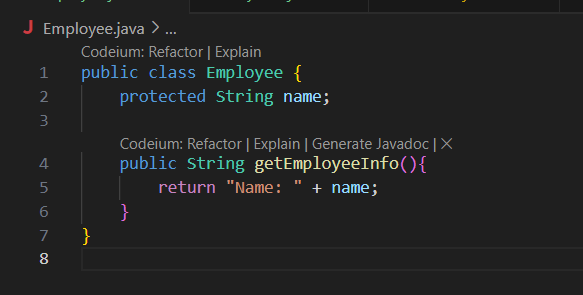
Link Github:

* **4. Experiment 1 – Base form of Polymorphism**

-Electricity Bill:



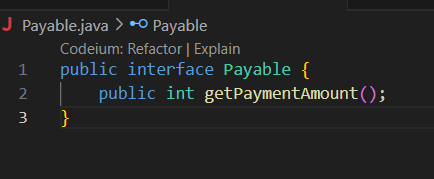
-Employee:

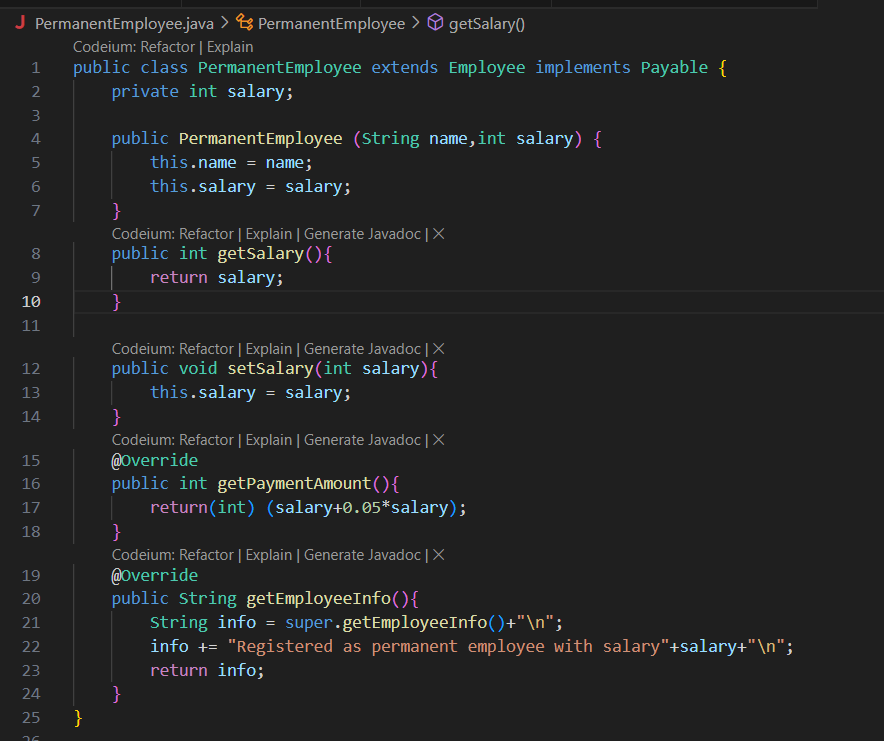


-Internship Employee

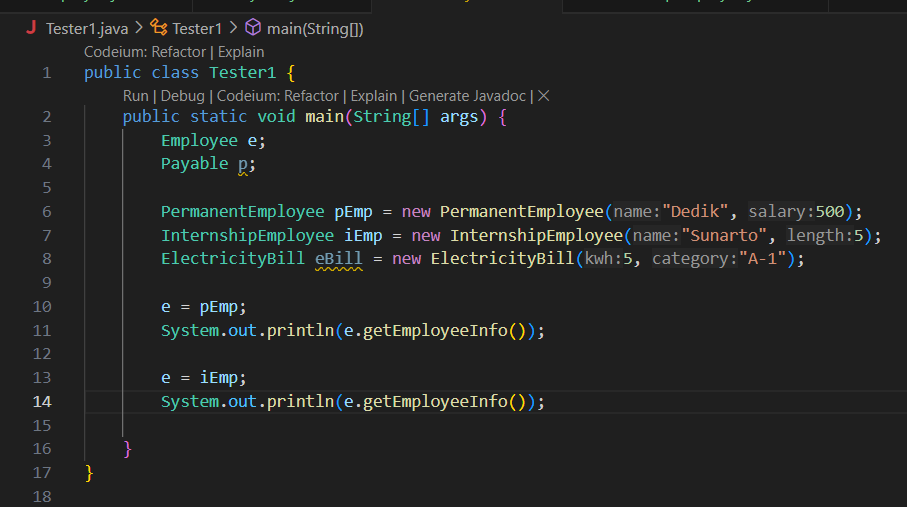


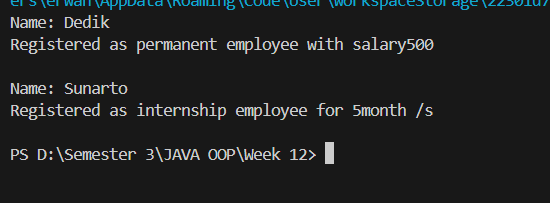
-payable:



-PermanentEmployee:  


-Tester1:



-Output:  


4.2. Question

1. Which classes are derived from Employee?   
-InternshipEmployee, PermanentEmployee

2. Which classes are implemented interface Payable?

- ElectricityBill

- PermanentEmployee

Both of these classes implement the getPaymentAmount() method defined in the Payable interface.

3. In class Tester1, look at line-10 and 11. Why we can assign e with object pEmp (an object from class PermanentEmployee) and object iEmp ( an object from class InternshipEmploye) ?

-Inheritance: PermanentEmployee and InternshipEmployee both extend Employee. This means they inherit its properties and methods and can be treated as instances of Employee.

Polymorphism: In Java, a variable of a superclass type (in this case, Employee) can hold a reference to an object of any of its subclasses. So, Employee e can point to an instance of PermanentEmployee or InternshipEmployee.

When you assign e = pEmp; and e = iEmp;, you’re effectively treating pEmp and iEmp as instances of their superclass (Employee), which allows for flexible and reusable code. You can call methods defined in Employee (like getEmployeeInfo()) without needing to know the exact subclass at compile time.

4. In class Tester1, look at line 12 and 13. Why we can assign p, with object pEmp (an object from class PermanentEmployee) and object eBill (an object from class ElectricityBill) ?

-In Tester1, the variable p is of type Payable, and we can assign it to pEmp (an object of PermanentEmployee) and eBill (an object of ElectricityBill) because both PermanentEmployee and ElectricityBill implement the Payable interface.  
  
Interface Implementation: Both PermanentEmployee and ElectricityBill implement the Payable interface, meaning they provide an implementation for the getPaymentAmount() method defined in Payable. This allows instances of these classes to be referenced by a Payable variable.

Polymorphism: In Java, a variable of an interface type (in this case, Payable) can hold a reference to any object of a class that implements that interface. Therefore, Payable p can point to instances of PermanentEmployee and ElectricityBill.

When you assign p = pEmp; and p = eBill;, you are treating pEmp and eBill as Payable instances, which allows you to call methods defined in the Payable interface (like getPaymentAmount()) without knowing the specific class implementing Payable at compile time. This flexibility makes the code more modular and extendable

5. Add this code to line 14 and 15:

p = iEmp;

e = eBill;

Why it resulted error?

-When you attempt to add the following lines to the Tester1 class:  
-p = iEmp;

Error: InternshipEmployee does not implement Payable.

Fix: Either have InternshipEmployee implement Payable or avoid assigning it to a Payable variable.

e = eBill;

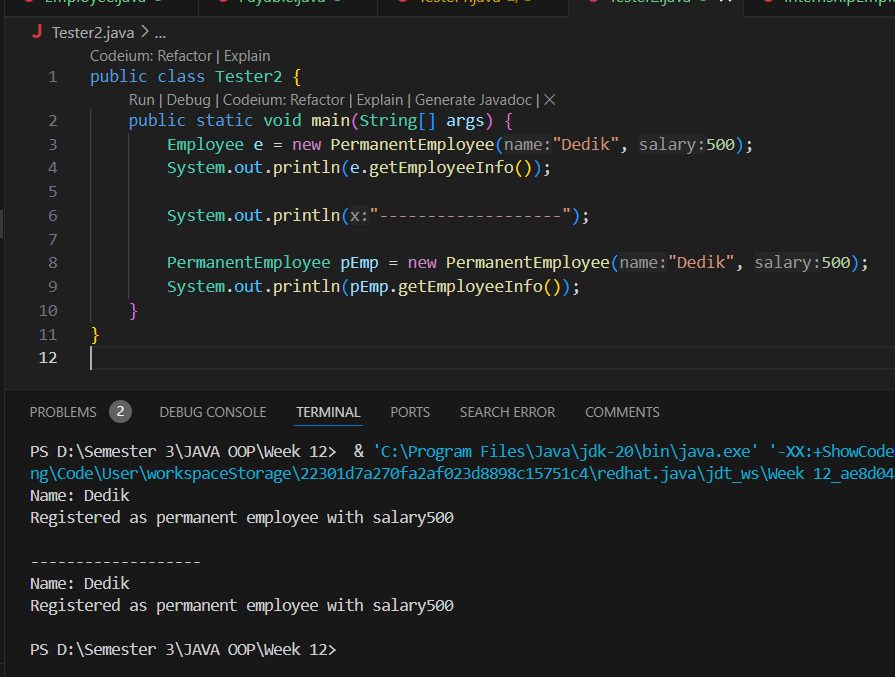
Error: ElectricityBill is not a subclass of Employee.

Fix: Use a compatible type like Payable for eBill or avoid this assignment altogether.

6. Make a conclusion about polymorphism concept

- Polymorphism in object-oriented programming is the ability of different classes to be treated as instances of the same superclass or interface. This allows a single variable or method to operate on objects of various types in a way that is consistent and predictable.

* **5. Experiment 2 – Virtual method invocation**



5.2. Question

1. Look at class Tester2, why method calling e.getEmployeeInfo() at line 8 and pEmp.getEmployeeInfo() at line 10 has the same result?

-In Tester2, the method calls e.getEmployeeInfo() on line 8 and pEmp.getEmployeeInfo() on line 10 produce the same result because both e and pEmp refer to instances of the PermanentEmployee class.

-Since both e and pEmp refer to PermanentEmployee objects, calling getEmployeeInfo() on either variable produces the same result, even though e is of type Employee. This is a demonstration of polymorphism where the actual method that gets executed depends on the object’s type at runtime, not on the reference type.

2. Why method calling e.getEmployeeInfo() called virtual method invocation, but not pEmp.getEmployeeInfo()?

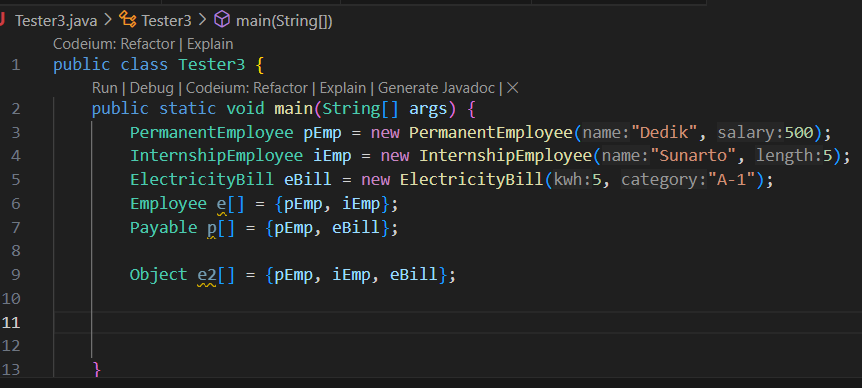
- The method call e.getEmployeeInfo() is considered a virtual method invocation because e is a reference of the superclass type Employee, but it points to an instance of the subclass PermanentEmployee. In Java, when a method is called on a superclass reference that could potentially point to different subclass objects, the JVM uses dynamic method dispatch to determine which version of the method (from which subclass) should be executed at runtime. This is what makes it "virtual"—the method invoked is determined dynamically, based on the actual object type.

3. So, whats the meaning about virtual method invocation? Why it called ‘virtual’?

- The term "virtual" originates from virtual functions in C++ and other early object-oriented languages. In these languages, a "virtual" function or method is one that can be overridden in derived classes and supports late binding (runtime determination of the method to be called). In Java, all instance methods (non-static methods) are virtual by default—meaning they can be overridden and are resolved at runtime based on the actual object type, rather than the declared type of the reference variable.

* **6. Experiment 3 – Heterogenous Collection**

**6. 1. Steps**

****

6.2. Question

1. Why array e at line 8, can be filled with different object type?

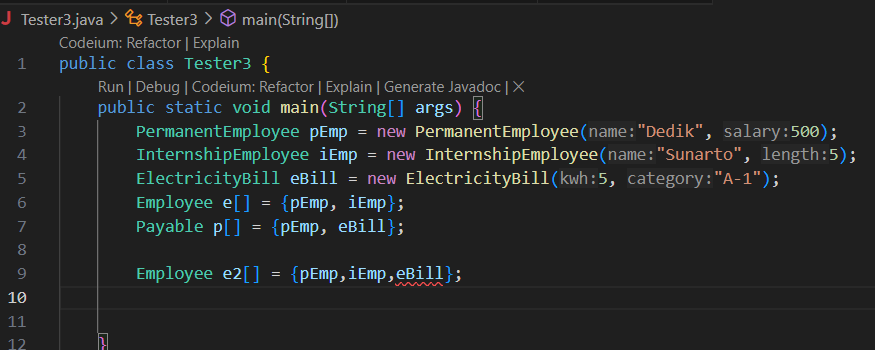
-The reason the array e at line 8 in your Tester3 class can be filled with different object types (pEmp of type PermanentEmployee and iEmp of type InternshipEmployee) is because both PermanentEmployee and InternshipEmployee are subclasses of Employee. In Java, arrays can store objects of the same type or subclasses of that type.

2. Why array p at line 9, can be filled with different object type?

- The array p at line 9 in your Tester3 class can be filled with different object types because both pEmp (an instance of PermanentEmployee) and eBill (an instance of ElectricityBill) implement the same interface, namely the Payable interface

3. Look at line 10, why it returned an error?

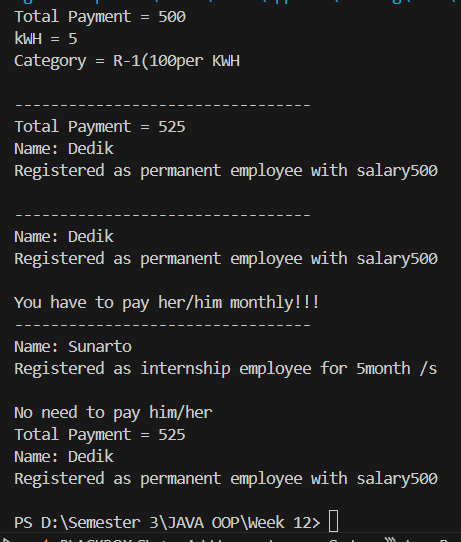
- the error occurs because eBill is an instance of the class ElectricityBill, which does not extend Employee. Therefore, it cannot be added to an array of type Employee[].



* **Experiment 4 – Polymorphism argument, instanceof and object casting**







7.2. Question

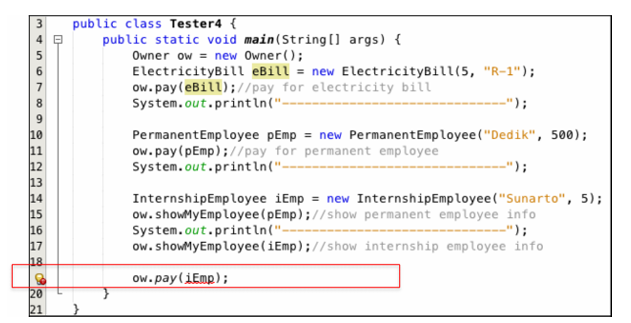
1. At class Tester4, in line 7 and 11, why we can call method ow.pay(eBill) and ow.pay(pEmp), eventhough pay() at class Owner has an argument type Payable?

- in the Tester4 class, the method calls ow.pay(eBill) and ow.pay(pEmp) work, even though the pay() method in the Owner class has an argument type Payable, because both ElectricityBill and PermanentEmployee implement the Payable interface.

2. What is the purpose we make an argument typed Payable at method pay() on class Owner?

- The purpose of making the argument type Payable in the pay() method of the Owner class is to generalize the method so that it can work with any object that implements the Payable interface. This enables the method to handle a wide variety of objects that have a common behavior (i.e., they can be paid) but might be of different types, such as ElectricityBill, PermanentEmployee, or any other class that implements the Payable interface

3. Add this line of code:



Why error happened?

-Because The method pay(Payable p) expects an argument that implements the Payable interface, but InternshipEmployee does not implement Payable. Therefore, the compiler will reject the call to ow.pay(iEmp) because it cannot automatically convert an InternshipEmployee object to a Payable object.

4. Look at class Owner, is it important we add p instanceof ElectricityBill at line 6 ?

- it is important if you want to handle the specific behavior of ElectricityBill objects separately from other Payable objects in the pay() method. The purpose of using instanceof is to perform type checking so that you can execute code tailored to ElectricityBill objects

5. Is still needed to cast this object (ElectricityBill eb = (ElectricityBill) p)? Why object p typed Payable must be casted into object eb?

- In the current design, casting is still required because:

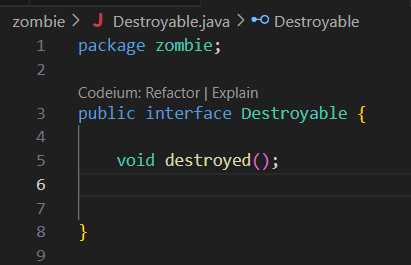
Payable doesn’t define all the methods that are available in ElectricityBill.

You need to access ElectricityBill-specific methods (getBillInfo()) that aren't part of the Payable interface.

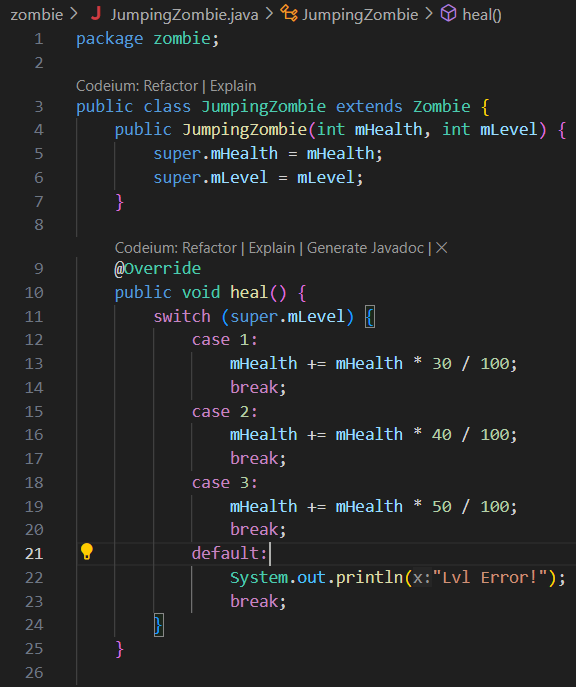
**8. Assignment**

**Barrier:  
  
**

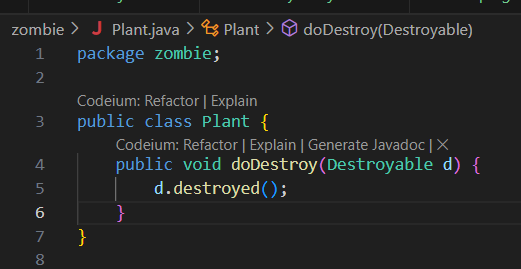
**Destroyable**

****

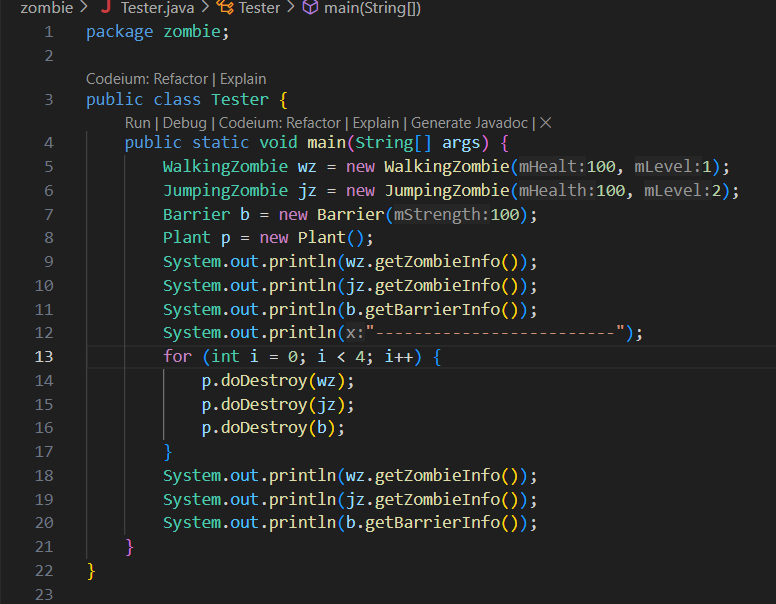
**JumpingZombie:**

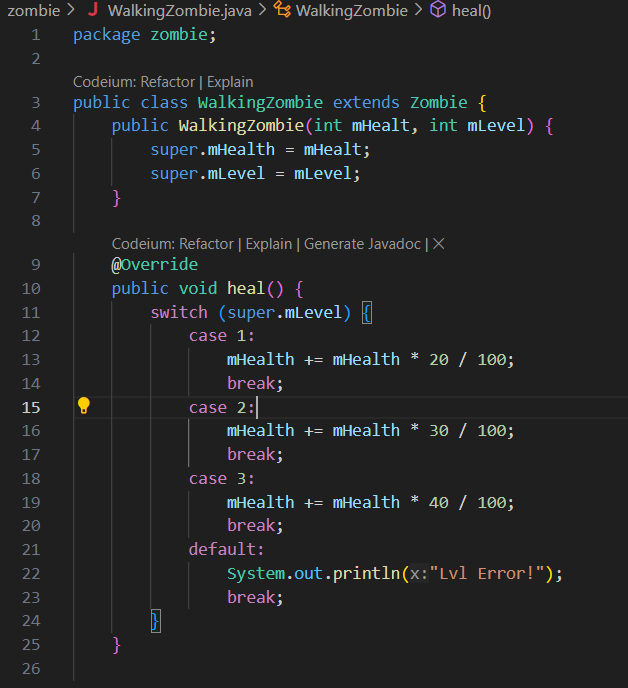
****

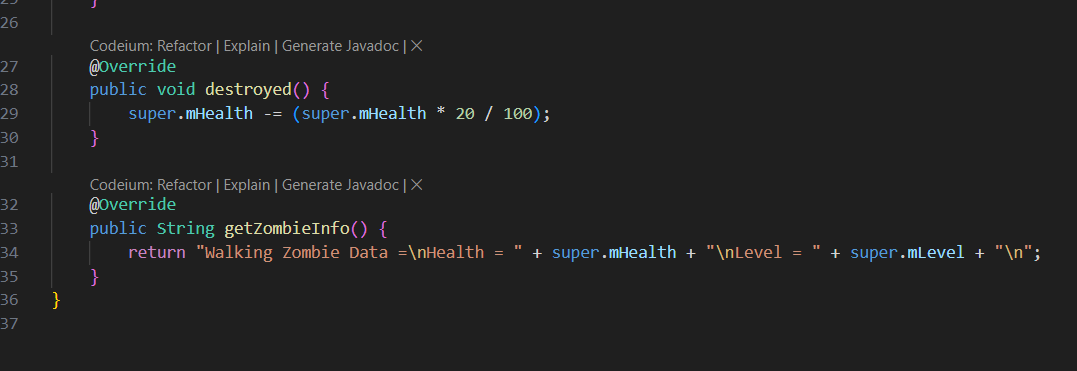
**Plant:**

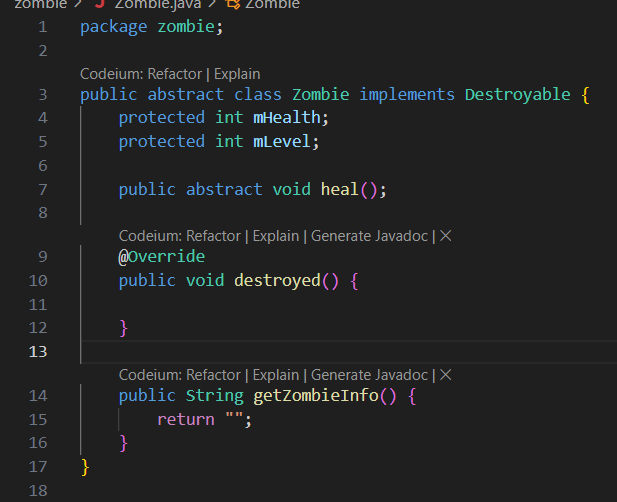
****

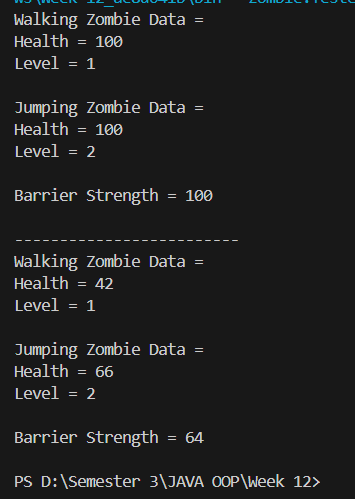
**Tester:**

****

**WalkingZombie:  
**

****

**Zombie:  
**

**Output:  
**