JOBSHEET
POLIMORFISME

1. Kompetensi

Setelah melakukan percobaan pada jobsheet ini, diharapkan mahasiswa mampu:

a. Memahami konsep dan bentuk dasar polimorfisme
b. Memahami konsep virtual method invication
c. Menerapkan polimorfisme pada pembuatan heterogeneous collection
d. Menerapkan polimorfisme pada parameter/argument method
e. Menerapkan object casting untuk meng-ubah bentuk objek

2. Pendahuluan

Polimorfisme merupakan kemampuan suatu objek untuk memiliki banyak bentuk. Penggunaan polimorfisme yang paling umum dalam OOP terjadi ketika ada referensi super class yang digunakan untuk merujuk ke objek dari sub class. Dengan kata lain, ketika ada suatu objek yang dideklarasikan dari super class, maka objek tersebut bisa diinstansiasi sebagai objek dari sub class. Dari uraian tersebut bisa dilihat bahwa konsep polimorfisme bisa diterapkan pada class-class yang memiliki relasi inheritance (relasi generalisasi atau IS-A).

Selain pada class-class yang memiliki relasi inheritance, polimorfisme juga bisa diterapkan pada interface. Ketika ada objek yang dideklarasikan dari suatu interface, maka ia bisa digunakan untuk mereferensi ke objek dari class-class yang implements ke interface tersebut.

Untuk mengilustrasikan uraian di atas, diberikan contoh sebagai berikut ini. Terdapat interface **`Vegetarian`**, dan super class **`Animal`**. Kemudian dibuat class **`Deer`** yang merupakan sub-class dari **`Animal`** dan implements ke **`Vegetarian`**. Sedangkan class **`Lion`** sub-class dari **`Animal`**, dan tidak implements ke **`Vegetarian`**.

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
public class Lion extends Animal {}
```

Dari deklarasi class di atas, contoh deklarasi objek di bawah ini akan menunjukan mana yang valid dan yang tidak valid berdasarkan konsep polimorfisme.

```
Deer d = new Deer();
Lion l = new Lion();
```

```
Animal a = d;
Animal a2 = l;
Vegetarian v = d;
```

valid

```
Vegetarian v2 = l;
```

tidak valid

Dari contoh di atas, **a** (bertipe **Animal**) bisa digunakan untuk mereferensi ke objek **d** (merupakan objek dari **Deer**), karena class **Deer** merupakan turunan dari **Animal**. Demikian juga **a2** (bertipe **Animal**), juga bisa digunakan untuk mereferensi ke objek **l** (merupakan objek dari **Lion**), karena class **Lion** merupakan turunan dari **Animal**. Objek **v** (dideklarasikan dari interface **Vegetarian**) bisa juga digunakan untuk mereferensi ke objek **d** (objek dari class **Deer**), hal ini bisa dilakukan karena class **Deer** implements ke interface **Vegetarian**.

Sedangkan objek **v2** (didekarasikan dari interface **Vegetarian**) **tidak bisa** digunakan untuk mereferensi objek **l** (objek dari class **Lion**), karena class **Lion** **tidak implements** ke interface **Vegetarian**. Ilustrasi tersebut bisa menunjukkan konsep dan bentuk dasar dari polimorfisme.

Virtual method Invocation

Virtual method invocation terjadi ketika ada pemanggilan overriding method dari suatu objek polimorfisme. Disebut virtual karena antara method yang dikenali oleh compiler dan method yang dijalankan oleh JVM berbeda.

```
public class Animal{
      public void walk(){
            System.out.println("The animal is walking around
            the jungle");
      }
```

```
}

public class Deer extends Animal {
      @override
      public void walk(){
            System.out.println("The deer is walking around
            the jungle");
}
```

Ketika ada suatu objek polimorfisme **a**, misalkan:

```
Deer d = new Deer();
Animal a = d;
```

Kemudian dipanggil method overriding darinya, maka saat itu terjadi pemanggilan method virtual, seperti:

```
a.walk();
```

Saat compile time, compiler akan mengenali method **walk()** yang akan dipanggil adalah method **walk()** yang ada di class **Animal**, karena objek **a** bertipe **Animal**. Tetapi saat dijalankan (run time), maka yang dijalankan oleh JVM justru method **walk()** yang ada di class **Deer**. Akan berbeda halnya jika pemanggilan method walk() dilakukan dari objek **d** (bukan objek polimorfisme), seperti

```
d.walk();
```

maka method **walk()** yang dikenali saat compile time oleh compiler dan yang dijalankan saat runtime oleh JVM adalah sama-sama method **walk()** yang ada di class **Deer** (karena objek **d** dideklarasikan dari class **Deer**).

Heterogeneous Collection

Dengan adanya konsep polimorfisme, maka variabel array bisa dibuat heterogen. Artinya di dalam array tersebut bisa berisi berbagai macam objek yang berbeda. Contoh:

```
Animal animalArray[] = new Animal[2];
animalArray[0] = new Deer();
animalArray[1] = new Lion();
```

Dari contoh tersebut data pertama dari array **animalArray** berisi objek **Deer**, dan data kedua dari **animalArray** berisi objek **Lion**. Hal ini bisa dilakukan

karena array **animalArray** dideklarasikan dari class **Animal** (superclass dari **Deer** dan **Lion**).

Polymorphic Argument

Polimorfisme juga bisa diterapkan pada argument suatu method. Tujuannya agar method tersebut bisa menerima nilai argument dari berbagai bentuk objek. Misalkan dibuat class baru sebagai berikut:

```
public class Human{
      public void drive(Animal anim){
            anim.walk();
       }
}
```

Perhatikan method **drive()**, ia memiliki argument berupa **Animal**. Karena **Animal** memiliki subclass **Lion** dan **Deer**, maka method **drive()** tersebut akan bisa menerima argument berupa objek dari **Deer** maupun objek dari **Lion**.

```
Deer d = new Deer();
Lion l = new Lion();
Human hum = new Human();
hum.drive(d);
hum.drive(l);
```

Operator instanceof

Operator instanceof bisa digunakan untuk mengecek apakah suatu objek merupakan hasil instansiasi dari suatu class tertentu. Hasil dari instanceof berupa nilai boolean. Misalkan dibuat objek d dan l.

```
Deer d = new Deer();
Lion l = new Lion();
Animal a1 = d;
Animal a2 = l;
```

Jika kemudian digunakan operator instanceof, misalkan

```
a1 instanceof Deer  →→ akan menghasilkan true
a2 instanceof Lion →→ akan menghasilkan true
```

Object Casting

Casting objek digunakan untuk mengubah tipe dari suatu objek. Jika ada suatu objek dari subclass kemudian tipenya diubah ke superclass, maka hal ini termasuk ke upcasting. Contoh:

```
Deer d = new Deer();
Animal a1 = d; // proses ini bisa disebut juga upcasting
```

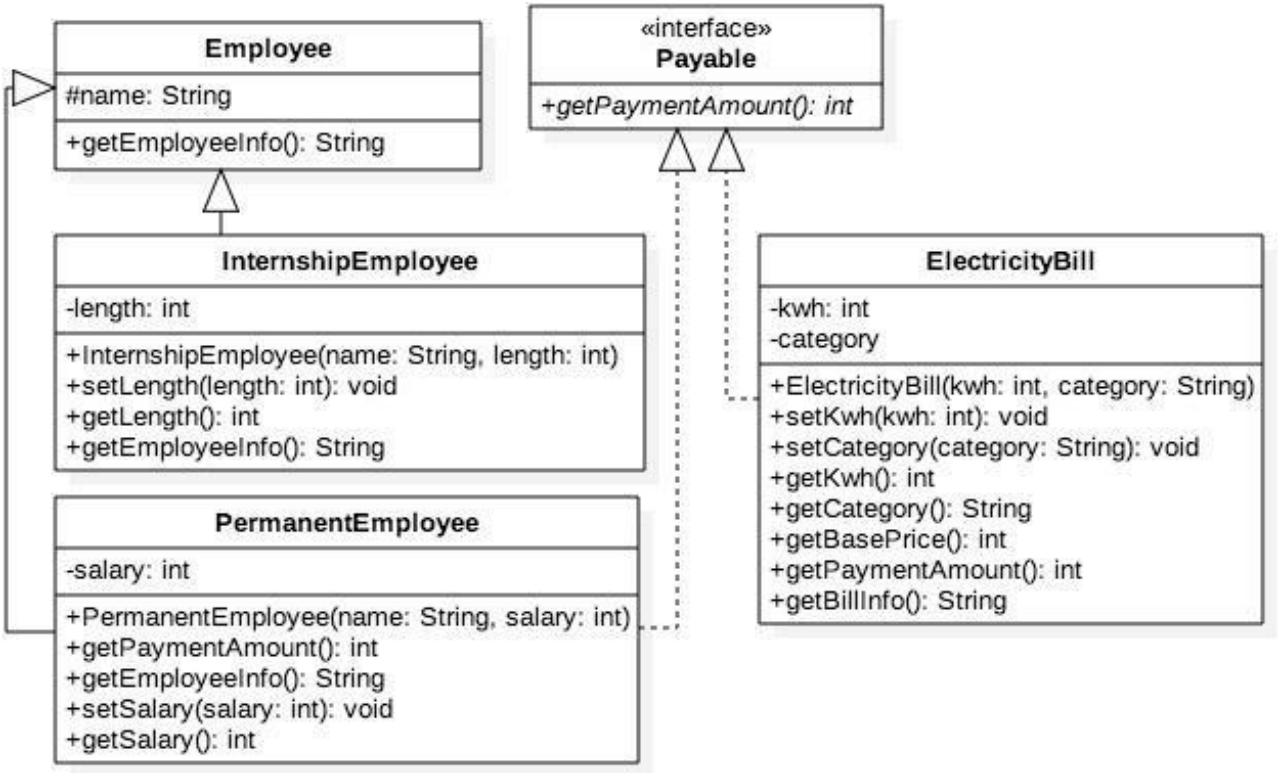
Downcast terjadi jika ada suatu objek superclass, kemudian diubah menjadi objek dari subclass. Contoh:

```
Deer  d = new Deer();
 Animal a1 = d; // proses ini bisa disebut juga upcasting
Deer  d = (Deer) a1; //proses downcasting
```

Proses downcasting sering disebut juga sebagai explicit casting, karena bentuk tujuan dari casting harus dituliskan dalam tanda kurung, di depan objek yang akan di-casting.

3. Studi Kasus

Untuk percobaan pada jobsheet ini akan digunakan class diagram di bawah ini:



Dalam suatu perusahaan, pemilik pada tiap bulannya harus membayar gaji pegawai tetap dan rekening listrik. Selain pegawai tetap perusahaan juga memiliki pegawai magang, dimana pegawai ini tidak mendapatkan gaji.

4. Percobaan 1 – Bentuk dasar polimorfisme

4.1. Langkah Percobaan

1. Buat class **`Employee`**

```
public class Employee {
    protected String name;

    public String getEmployeeInfo(){
        return "Name = "+name;
    }
}
```

2. Buat interface **`Payable`**

```
public interface Payable {
    public int getPaymentAmount();
}
```

3. Buat class **`InternshipEmployee`**, subclass dari **`Employee`**

```
public class InternshipEmployee extends Employee{
    private int length;

    public InternshipEmployee(String name, int length) {
        this.length = length;
        this.name = name;
    }
    public int getLength() {
        return length;
    }
    public void setLength(int length) {
        this.length = length;
    }
    @Override
    public String getEmployeeInfo(){
        String info = super.getEmployeeInfo()+"\n";
        info += "Registered as internship employee for "+length+" month/s\n";
        return info;
    }
}
```

4. Buat class **`PermanentEmployee`**, subclass dari **`Employee`** dan implements ke **`Payable`**

```
3   public class PermanentEmployee extends Employee implements Payable{
4       private int salary;
5
6       public PermanentEmployee(String name, int salary) {
7           this.name = name;
8           this.salary = salary;
9       }
10      public int getSalary() {
11          return salary;
12      }
13      public void setSalary(int salary) {
14          this.salary = salary;
15      }
16      @Override
        public int getPaymentAmount() {
18          return (int) (salary+0.05*salary);
19      }
20      @Override
        public String getEmployeeInfo(){
22          String info = super.getEmployeeInfo()+"\n";
23          info += "Registered as permanent employee with salary "+salary+"\n";
24          return info;
25      }
26  }
```

5. Buat class **`ElectricityBill`** yang implements ke interface **`PayableS`**

```
3   public class ElectricityBill implements Payable{
4       private int kwh;
5       private String category;
6
7       public ElectricityBill(int kwh, String category) {
8           this.kwh = kwh;
9           this.category = category;
10      }
11      public int getKwh() {
12          return kwh;
13      }
14      public void setKwh(int kwh) {
15          this.kwh = kwh;
16      }
17      public String getCategory() {
18          return category;
19      }
20      public void setCategory(String category) {
21          this.category = category;
22      }
23      @Override
25      public int getPaymentAmount() {
26          return kwh*getBasePrice();
27      }
28      public int getBasePrice(){
29          int bPrice = 0;
30          switch(category){
31              case "R-1" : bPrice = 100;break;
32              case "R-2" : bPrice = 200;break;
33          }
34          return bPrice;
35      }
36      public String getBillInfo(){
37          return "kWH = "+kwh+"\n"+
38                  "Category = "+category+"("+getBasePrice()+" per kWH)\n";
39      }
  }
```

6. Buat class **`Tester1`**

```
3   public class Tester1 {
4       public static void main(String[] args) {
5           PermanentEmployee pEmp = new PermanentEmployee("Dedik", 500);
6           InternshipEmployee iEmp = new InternshipEmployee("Sunarto", 5);
7           ElectricityBill eBill = new ElectricityBill(5, "A-1");
8           Employee e;
9           Payable p;
10          e = pEmp;
11          e = iEmp;
12          p = pEmp;
13          p = eBill;
14      }
15  }
```

4.2. Pertanyaan

1. Class apa sajakah yang merupakan turunan dari class **`Employee`**?

   = **IntershipEmployee dan PermanentEmployee**

2. Class apa sajakah yang implements ke interface **`Payable`**?

   = **PermanentEmployee dan ElectricityBill**

3. Perhatikan class **`Tester1`**, baris ke-10 dan 11. Mengapa **`e`**, bisa diisi dengan objek **`pEmp`** (merupakan objek dari class **`PermanentEmployee`**) dan objek **`iEmp`** (merupakan objek dari class **`InternshipEmploye`**) ?

   = Variabel e dapat diisi dengan objek pEmp dan iEmp karena PermanentEmployee dan InternshipEmployee merupakan subkelas dari Employee (hubungan pewarisan).

4. Perhatikan class **`Tester1`**, baris ke-12 dan 13. Mengapa **`p`**, bisa diisi denganobjek**`pEmp`** (merupakan objek dari class **`PermanentEmployee`**) dan objek **`eBill`** (merupakan objek dari class **`ElectricityBill`**) ?

   = Variabel p dapat diisi dengan objek pEmp dan eBill karena baik PermanentEmployee dan ElectricityBill mengimplementasikan antarmuka Payable.

5. Coba tambahkan sintaks:

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

   pada baris 14 dan 15 (baris terakhir dalam method **`main`**) ! Apa yang menyebabkan error?

   = Error disebabkan oleh InternshipEmployee tidak mengimplementasikan Payable interface ElectricityBillbukan merupakan subkelas dari Employee

6. Ambil kesimpulan tentang konsep/bentuk dasar polimorfisme!
   - **Sebuah objek dapat diperlakukan sebagai jenisnya sendiri atau jenis induknya**
   - **Variabel tipe antarmuka dapat mereferensikan objek apa pun yang mengimplementasikan antarmuka tersebut**
   - **Hal ini memungkinkan kode yang fleksibel dan dapat digunakan kembali melalui hubungan “is-a” yang ditetapkan oleh pewarisan dan implementasi antarmuka**

5. Percobaan 2 – Virtual method invocation

5.1. Langkah Percobaan

1. Pada percobaan ini masih akan digunakan class-class dan interface yang digunakan pada percobaan sebelumnya.
2. Buat class baru dengan nama **`Tester2`**.

```
3    public class Tester2 {
4        public static void main(String[] args) {
5            PermanentEmployee pEmp = new PermanentEmployee("Dedik", 500);
6            Employee e;
7            e = pEmp;
8            System.out.println(""+e.getEmployeeInfo());
9            System.out.println("------------------");
10           System.out.println(""+pEmp.getEmployeeInfo());
11       }
12   }
```

```
public class Tester2 {
    public static void main(String[] args) {
        PermanentEmployee pEmp = new
        PermanentEmployee("Dedik", 500);
        Employee e;
        e = pEmp;
        System.out.println(""+e.getEmployeeInfo());
        System.out.println("------------------------");

        System.out.println(""+pEmp.getEmployeeInfo());
    }
}
```

3. Jalankan class **`Tester2`**, dan akan didapatkan hasil sebagai berikut:

```
run:
Name = Dedik
Registered as permanent employee with salary 500

------------------
Name = Dedik
Registered as permanent employee with salary 500
```

```
Name = Dedik
Registered as permanent employee with salary 500

------------------------
Name = Dedik
Registered as permanent employee with salary 500

BUILD SUCCESSFUL (total time: 0 seconds)
```

5.2. Pertanyaan

1. Perhatikan class **`Tester2`** di atas, mengapa pemanggilan

`e.getEmployeeInfo()` pada baris 8 dan `pEmp.getEmployeeInfo()` pada baris 10 menghasilkan hasil sama?

= Baik e.getEmployeeInfo( ) maupun pEmp.getEmployeeInfo() menghasilkan hasil yang sama karena e mereferensikan objek PermanentEmployee yang sama dengan pEmp, dan karena penimpaan metode, maka metode kelas turunannya yang dipanggil.

2. Mengapa pemanggilan method `e.getEmployeeInfo()` disebut sebagai pemanggilan method virtual (virtual method invication), sedangkan `pEmp.getEmployeeInfo()` tidak?

   = e. getEmployeeInfo() disebut pemanggilan metode virtual karena pada saat kompilasi, kompilator melihat e sebagai tipe Employee, tetapi pada saat runtime JVM memanggil metode versi PermanentEmployee. pEmp.getEmployeeInfo() tidak bersifat virtual karena kompilator dan JVM mengetahui bahwa ia memanggil metode PermanentEmployee.

3. Jadi apakah yang dimaksud dari virtual method invocation? Mengapa disebut virtual?

   = virtual method invocation adalah ketika pemanggilan metode diselesaikan pada saat runtime, bukan pada saat kompilasi. Disebut "virtual" karena metode yang sebenarnya dipanggil tidak ditentukan hingga saat runtime, berdasarkan tipe objek yang sebenarnya, bukan tipe referensi.

6. Percobaan 3 – Heterogenous Collection

6.1. Langkah Percobaan

1. Pada percobaan ke-3 ini, masih akan digunakan class-class dan interface pada percobaan sebelumnya.
2. Buat class baru **`Tester3`**.

```
3   public class Tester3 {
4       public static void main(String[] args) {
5           PermanentEmployee pEmp = new PermanentEmployee("Dedik", 500);
6           InternshipEmployee iEmp = new InternshipEmployee("Sunarto", 5);
7           ElectricityBill eBill = new ElectricityBill(5, "A-1");
            Employee e[] = {pEmp, iEmp};
            Payable p[] = {pEmp, eBill};
            Employee e2[] = {pEmp, iEmp, eBill};
11      }
12  }
```

```
public class Tester3 {
    public static void main(String[] args) {
        PermanentEmployee pEmp = new
        PermanentEmployee("Dedik", 500);
        InternshipEmployee iEmp = new
        InternshipEmployee("Sunarto", 5);
        ElectricityBill eBill = new
        ElectricityBill(5, "A-1");

        Employee[] e = {pEmp, iEmp};
        Payable[] p = {pEmp, eBill};
        Employee[] e2 = {pEmp, iEmp, eBill};
    }
}
```

6.2. Pertanyaan

1. Perhatikan array **`e`** pada baris ke-8, mengapa ia bisa diisi dengan objek-objek dengan tipe yang berbeda, yaitu objek **`pEmp`** (objek dari **`PermanentEmployee`**) dan objek **`iEmp`** (objek dari **`InternshipEmployee`**) ?

   = Array e bisa berisi pEmp dan iEmp karena baik PermanentEmployee dan InternshipEmployee adalah subkelas dari Employee.

2. Perhatikan juga baris ke-9, mengapa array **`p`** juga biisi dengan objek-objek dengan tipe yang berbeda, yaitu objek **`pEmp`** (objek dari **`PermanentEmployee`**) dan objek **`eBill`** (objek dari **`ElectricityBilling`**) ?

= Array p bisa berisi pEmp dan eBill karena baik

PermanentEmployee maupun ElectricityBill mengimplementasikan

Payable Interface.

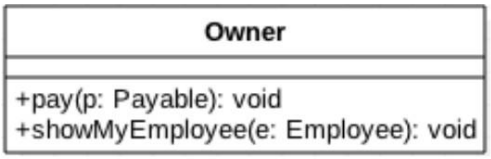
3. Perhatikan baris ke-10, mengapa terjadi error?
   = Kesalahan terjadi karena eBill (ElectricityBill) bukan merupakan subkelas dari Employee, sehingga tidak dapat disimpan dalam array Employee.

7. Percobaan 4 – Argumen polimorfisme, instanceof dan casting objek

7.1. Langkah Percobaan

1. Percobaan 4 ini juga masih menggunakan class-class dan interface yang digunakan pada percobaan sebelumnya.

4. Buat class baru dengan nama **`Owner`**. **`Owner`** bisa melakukan pembayaran baik kepada pegawai permanen maupun rekening listrik melalui method **`pay()`**. Selain itu juga bisa menampilkan info pegawai permanen maupun pegawai magang melalui method **`showMyEmployee()`**.




```
3   public class Owner {
4       public void pay(Payable p){
5           System.out.println("Total payment = "+p.getPaymentAmount());
6           if(p instanceof ElectricityBill){
7               ElectricityBill eb = (ElectricityBill) p;
8               System.out.println(""+eb.getBillInfo());
9           }else if(p instanceof PermanentEmployee){
10              PermanentEmployee pe = (PermanentEmployee) p;
11              pe.getEmployeeInfo();
12              System.out.println(""+pe.getEmployeeInfo());
13          }
14      }
15      public void showMyEmployee(Employee e){
16          System.out.println(""+e.getEmployeeInfo());
17          if(e instanceof PermanentEmployee)
18              System.out.println("You have to pay her/him monthly!!!");
19          else
20              System.out.println("No need to pay him/her :)");
21      }
22  }
```

```java
public class Owner {
    public void pay(Payable p) {
    System.out.println("Total payment = " +
    p.getPaymentAmount());
    if(p instanceof ElectricityBill) {
        ElectricityBill eb = (ElectricityBill) p;
        System.out.println("" +
        eb.getBillInfo());
    } else if(p instanceof PermanentEmployee) {
        PermanentEmployee pe = (PermanentEmployee) p;
        pe.getEmployeeInfo();
        System.out.println("" + pe.getEmployeeInfo());
    }
    }

    public void showMyEmployee(Employee e) {
        System.out.println("" + e.getEmployeeInfo());
        if(e instanceof PermanentEmployee) {
            System.out.println("You have to pay her/him monthly!!!");
        } else {
            System.out.println("No need to pay him/her :)");
        }
    }
}
```

2. Buat class baru **`Tester4`**.

```java
public class Tester4 {
    public static void main(String[] args) {
        Owner ow = new Owner();
        ElectricityBill eBill = new ElectricityBill(5, "R-1");
        ow.pay(eBill);//pay for electricity bill
        System.out.println("------------------------------");

        PermanentEmployee pEmp = new PermanentEmployee("Dedik", 500);
        ow.pay(pEmp);//pay for permanent employee
        System.out.println("------------------------------");

        InternshipEmployee iEmp = new InternshipEmployee("Sunarto", 5);
        ow.showMyEmployee(pEmp);//show permanent employee info
        System.out.println("------------------------------");
        ow.showMyEmployee(iEmp);//show internship employee info
    }
}
```

3. Jalankan class **`Tester4`**, dan akan didaptkan hasil sebagai berikut:

```
run:
Total payment = 1000
kWH = 5
Category = R-1(200 per kWH)

------------------------------
Total payment = 525
Name = Dedik
Registered as permanent employee with salary 500

------------------------------
Name = Dedik
Registered as permanent employee with salary 500

You have to pay her/him monthly!!!
------------------------------
Name = Sunarto
Registered as internship employee for 5 month/s

No need to pay him/her :)
```

```java
public class Tester4 {
    public static void main(String[] args) {
        Owner owner = new Owner();
        ElectricityBill eBill = new
        ElectricityBill(5, "R-1");
        PermanentEmployee pEmp = new
        PermanentEmployee("Dedik", 500);
        InternshipEmployee iEmp = new
        InternshipEmployee("Sunarto", 5);

        owner.pay(eBill);
        owner.pay(pEmp);
        owner.showMyEmployee(pEmp);
        owner.showMyEmployee(iEmp);
    }
}
```

Hasil:

```
Total payment = 0
kWH = 5
 Category = R-1(0 per kWH)

Total payment = 500
Name = Dedik
Registered as permanent employee with salary 500

Name = Dedik
Registered as permanent employee with salary 500

You have to pay her/him monthly!!!
Name = Sunarto
Registered as internship employee for 5 month/s

No need to pay him/her :)
BUILD SUCCESSFUL (total time: 0 seconds)
```

7.2. Pertanyaan

1. Perhatikan class **`Tester4`** baris ke-7 dan baris ke-11, mengapa pemanggilan **`ow.pay(eBill)`** dan **`ow.pay(pEmp)`** bisa dilakukan, padahal jika diperhatikan method **`pay()`** yang ada di dalam class **`Owner`** memiliki argument/parameter bertipe **`Payable`**?

Jika diperhatikan lebih detil eBill merupakan objek dari **`ElectricityBill`** dan pEmp merupakan objek dari **`PermanentEmployee`**?

= owner.pay(eBill) dan owner.pay(pEmp) berfungsi karena ElectricityBill dan PermanentEmployee mengimplementasikan Payable Interface, sehingga menjadikannya argumen yang valid untuk metode pay yang mengharapkan parameter Payable.

2. Jadi apakah tujuan membuat argument bertipe **`Payable`** pada method **`pay()`** yang ada di dalam class **`Owner`**?

   = Tujuan membuat tipe argumen metode pay() menjadi Payable adalah untuk mengizinkan metode tersebut memproses objek apa pun yang mengimplementasikan Payable Interface, membuat metode tersebut lebih fleksibel dan dapat digunakan kembali.

3. Coba pada baris terakhir method **`main()`** yang ada di dalam class **`Tester4`** ditambahkan perintah **`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  }
```

   Mengapa terjadi error?

   = Menambahkan owner.pay(iEmp) akan menyebabkan kesalahan karena InternshipEmployee tidak mengimplementasikan Payable Interface

4. Perhatikan class **`Owner`**, diperlukan untuk apakah sintaks **`p instanceof ElectricityBill`** pada baris ke-6 ?

   = Operator instanceof digunakan untuk memeriksa apakah objek Payable p secara spesifik merupakan objek ElectricityBill, sehingga memungkinkan pemrosesan tipe yang tepat.

5. Perhatikan kembali class Owner baris ke-7, untuk apakah casting objek disana (**`ElectricityBill eb = (ElectricityBill) p`**) diperlukan ? Mengapa objek **`p`** yang bertipe **`Payable`** harus di-casting ke dalam objek **`eb`** yang bertipe **`ElectricityBill ?`**

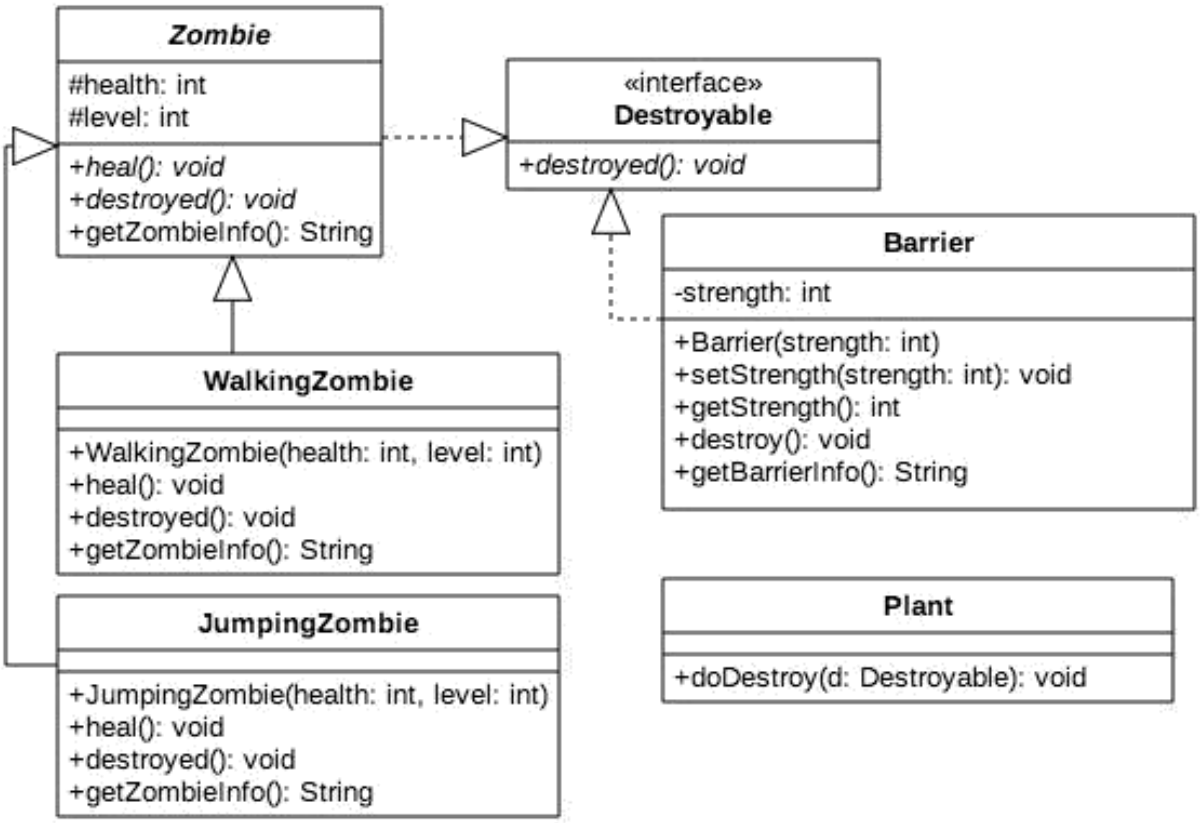
= Casting diperlukan karena meskipun p bertipe Payable, untuk mengakses metode-metode khusus ElectricityBill seperti getBillInfo(), kita perlu melakukan casting ke ElectricityBill. Hal ini dikenal sebagai downcasting, mengubah dari tipe Interface ke tipe kelas spesifik tertentu

8. Tugas

Dalam suatu permainan, Zombie dan Barrier bisa dihancurkan oleh Plant dan bisa menyembuhkan diri. Terdapat dua jenis Zombie, yaitu Walking Zombie dan Jumping Zombie. Kedua Zombie tersebut memiliki cara penyembuhan yang berbeda, demikian juga cara penghancurannya, yaitu ditentukan oleh aturan berikut ini:

- Pada WalkingZombie
    - Penyembuhan : Penyembuhan ditentukan berdasar level zombie yang bersangkutan
        - Jika zombie level 1, maka setiap kali penyembuhan, health akan bertambah 20%
        - Jika zombie level 2, maka setiap kali penyembuhan, health akan bertambah 30%
        - Jika zombie level 3, maka setiap kali penyembuhan, health akan bertambah 40%
    - Penghancuran : setiap kali penghancuran, health akan berkurang 2%
- Pada Jumping Zombie
    - Penyembuhan : Penyembuhan ditentukan berdasar level zombie yang bersangkutan
        - Jika zombie level 1, maka setiap kali penyembuhan, health akan bertambah 30%
        - Jika zombie level 2, maka setiap kali penyembuhan, health akan bertambah 40%
        - Jika zombie level 3, maka setiap kali penyembuhan, health akan bertambah 50%
    - Penghancuran : setiap kali penghancuran, health akan berkurang 1%

Buat program dari class diagram di bawah ini!



Contoh: jika class Tester seperti di bawah ini:

```
3   public class Tester {
4       public static void main(String[] args) {
5           WalkingZombie wz = new WalkingZombie(100, 1);
6           JumpingZombie jz = new JumpingZombie(100, 2);
7           Barrier b = new Barrier(100);
8           Plant p = new Plant();
9           System.out.println(""+wz.getZombieInfo());
10          System.out.println(""+jz.getZombieInfo());
11          System.out.println(""+b.getBarrierInfo());
12          System.out.println("----------------------");
13          for(int i=0;i<4;i++){//Destroy the enemies 4 times
14              p.doDestroy(wz);
15              p.doDestroy(jz);
16              p.doDestroy(b);
17          }
18          System.out.println(""+wz.getZombieInfo());
19          System.out.println(""+jz.getZombieInfo());
20          System.out.println(""+b.getBarrierInfo());
21      }
22  }
```

Akan menghasilkan output:

```
run:
Walking Zombie Data =
Health = 100
Level = 1

Jumping Zombie Data =
Health = 100
Level = 2

Barrier Strength = 100

----------------------
Walking Zombie Data =
Health = 42
Level = 1

Jumping Zombie Data =
Health = 66
Level = 2

Barrier Strength = 64

BUILD SUCCESSFUL (total time: 2 seconds)
```

a. **Destroyable**

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

b. **Zombie**

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

c. **Barrier**

```
public class Barrier implements Destroyable {
    private int strength;
    public Barrier(int strength) {
        this.strength = strength;
        }
    public void setStrength(int strength) {
        this.strength = strength;
    }
    public int getStrength() {
        return this.strength;
    }
    @Override
    public void destroyed() {
        this.strength -= (this.strength * 0.1);
    }
    public String getBarrierInfo() {
        return "\nBarrier Strength = " + this.strength;
    }

}
```

d. JumpingZombie

```java
public class JumpingZombie extends Zombie {
    public JumpingZombie(int health, int level) {
        this.health = health;
        this.level = level;
    }
    @Override
    public void heal() {
        switch(this.level) {
            case 1:
                this.health += (this.health * 0.3);
            break;
            case 2:
                this.health += (this.health * 0.4);
            break;
            case 3:
                this.health += (this.health * 0.5);
            break;
        }
    }
    @Override
    public void destroyed() {
        this.health -= (this.health * 0.01);
    }
    @Override
    public String getZombieInfo() {
        return "Jumping Zombie Data = " + super.getZombieInfo();
    }
}
```

e. WalkingZombie

```java
public class WalkingZombie extends Zombie {
    public WalkingZombie(int health, int level) {
        this.health = health;
        this.level = level;
    }
    @Override
    public void heal() {
        switch(this.level) {
            case 1:
                this.health += (this.health * 0.2);
            break;
            case 2:
                this.health += (this.health * 0.3);
            break;
            case 3:
                this.health += (this.health * 0.4);
            break;
        }
    }
    @Override
    public void destroyed() {
        this.health -= (this.health * 0.02);
    }
    @Override
    public String getZombieInfo() {
        return "Walking Zombie Data = " + super.getZombieInfo();
    }

}
```

f. Plant

```
public class Plant {
    public void doDestroy(Destroyable d) {
        if (d instanceof Zombie) {
            Zombie z = (Zombie) d;
            z.destroyed();
            System.out.println(z.getZombieInfo());
        } else if (d instanceof Barrier) {
            Barrier b = (Barrier) d;
            b.destroyed();
            System.out.println(b.getBarrierInfo());
        }
    }
}
```

g. Tester

```
public class Tester {
    public static void main(String[] args) {
        WalkingZombie wz = new WalkingZombie(100, 1);
        JumpingZombie jz = new JumpingZombie(100, 2);
        Barrier b = new Barrier(100);
        Plant p = new Plant();
        System.out.println("" + wz.getZombieInfo());
        System.out.println("" + jz.getZombieInfo());
        System.out.println("" + b.getBarrierInfo());
        System.out.println("\nPlant vs Zombie Battle Begin!");
        System.out.println("\nWave 1");
        p.doDestroy(wz);
        p.doDestroy(jz);
        p.doDestroy(b);
        System.out.println("\nWave 2");
        wz.heal();
        jz.heal();
        p.doDestroy(wz);
        p.doDestroy(jz);
        p.doDestroy(b);
    }
}
```

h. Hasil

```
Walking Zombie Data =
Health = 100
Level = 1
Jumping Zombie Data =
Health = 100
Level = 2

Barrier Strength = 100

Plant vs Zombie Battle Begin!

Wave 1
Walking Zombie Data =
Health = 98
Level = 1
Jumping Zombie Data =
Health = 99
Level = 2

Barrier Strength = 90

Wave 2
Walking Zombie Data =
Health = 114
Level = 1
Jumping Zombie Data =
Health = 136
Level = 2

Barrier Strength = 81
BUILD SUCCESSFUL (total time: 0 seconds)
```