Dasar-Dasar Pemrograman 2

Lab 07

Abstract Classes & Interfaces ♻

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

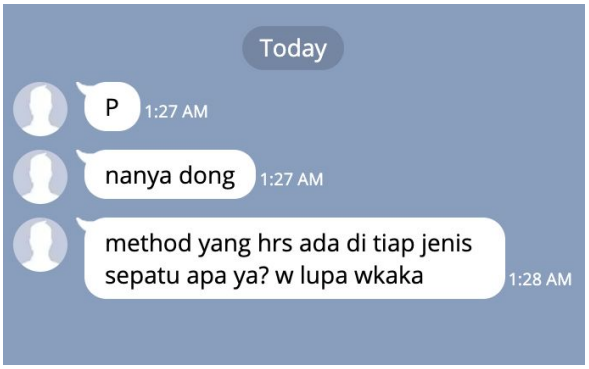
FAKULTAS ILMU KOMPUTER

Intermezzo

Think about this scenario, kalian sedang magang sebagai seorang software engineer di sebuah perusahaan penjual barang-barang *hypebeast* yang dimana sekarang sedang tahap pengimplementasian IT kedalam sistem mereka. Kalian sebagai *lead* dalam project ini ditugaskan untuk membuat sebuah program java untuk me-*manage* sepatu yang mereka punya. Karena sepatu jenisnya banyak, maka kalian dan tim diwajibkan untuk membuat class baru untuk setiap jenis sepatu. Setiap jenis sepatu memiliki paling tidak 3 method ini, **getColor(), getRetailPrice(),** dan **getResellPrice().**

Salah satu teman anda, memutuskan untuk membuat sebuah class **Sepatu** yang dimana nanti akan di-extend oleh tiap class jenis sepatu. Tetapi, pada saat pengimplementasian, salah satu teman anda yang lain lupa mengimplementasikan salah satu dari ketiga method diatas yang seharusnya diimplementasikan di setiap class jenis sepatu, sehingga anda memberitahunya, menegurnya, dan dia pun berkata tidak akan mengulanginya lagi.

2 minggu kemudian, saat anda sedang santai di pagi hari, teman anda mengirimkan pesan di Line:



Sejak saat inilah, anda berpikir bahwa sebenarnya *inheritance* saja tidak cukup untuk menyelesaikan masalah ini. Harus dibuat sebuah jenis class yang dimana merupakan sebuah template untuk class lain yang akan dibuat, supaya anda bisa yakin bahwa method tertentu harus diimplementasi. Disinlah anda memutuskan untuk belajar **Abstract Classes & Interfaces.**

Abstract Classes

Abstract classes adalah jenis class di Java yang berperan sebagai kerangka class lain yang ingin dibentuk. Abstract class **boleh memiliki abstract method dan boleh mengandung concrete method.**

Apa itu abstract method? Abstract methods adalah **method yang belum diimplementasikan di abstract class, tetapi wajib dioverride di class yang mengextend abstract class tersebut.**

Perhatikan contoh dibawah,

```
abstract class Sepatu {

   public abstract String getColor();

   public abstract int getResellPrice();

   public abstract int getRetailPrice();

   public void clean() {
       System.out.println("Swish Swish I'm clean!");
   }
}
```

Method **getcolor, getResellPrice dan getRetailPrice()** merupakan contoh dari **abstract methods**, mereka tidak diimplementasikan di abstract class, tetapi **wajib dioverride di class yang mengextend abstract class tersebut.**

Method **clean** merupakan **concrete method** karena method tersebut **tidak memiliki kata abstract** sehingga **tidak wajib dioverride di class yang mengextend abstract class tersebut.**

Sekarang kita akan membuat sebuah class yang mengextend, misalnya kita membuat sebuah class seperti ini:

```
class BataBoost700 extends Sepatu {
   private String color;
   private int resellPrice;
   private int retailPrice;
   public BataBoost700(String color, int resellPrice, int retailPrice) {
       this.color = color;
       this.resellPrice = resellPrice;
       this.retailPrice = retailPrice;
   }
}
```

Sebenarnya sekilas, tidak ada yang salah dengan class tersebut, tetapi bila kita mencoba compile, maka akan muncul error berikut:

```
Coba.java:19: error: BataBoost700 is not abstract and does not override
abstract method getRetailPrice() in Sepatu
```

Error ini menandakan bahwa class **BataBoost700** yang baru saja kita buat belum mengoverride abstract method yang ada di abstract class Sepatu.

Jadi cara kita men-solve nya adalah, karena semua class yang mengextend abstract class harus meng-override semua abstract method di abstract class, kita harus membuat method **getColor, getResellPrice,** dan **getRetailPrice.** Karena method **clean** merupakan concrete method (tidak ada kata abstract didepannya) maka dia tidak wajib dioverride di class yang mengextend.

```
class BataBoost700 extends Sepatu {
    private String color;
    private int resellPrice;
    private int retailPrice;

    public BataBoost700(String color, int resellPrice, int retailPrice) {
        this.color = color;
        this.resellPrice = resellPrice;
        this.retailPrice = retailPrice;
    }

    public String getColor() {
        return this.color;
    }

    public int getResellPrice() {
        return this.resellPrice;
    }

    public int getRetailPrice() {
        return this.retailPrice;
    }
}
```

Sekarang bila kita compile, maka dia tidak akan error.

Karena kita tahu sekarang bahwa ada method yang harus dan tidak harus diimplementasikan, maka kita bisa me-*refactor* code nya menjadi seperti ini:

```
abstract class Sepatu {
    public String color;
    public int resellPrice;
    public int retailPrice;

    public Sepatu(String color, int resellPrice, int retailPrice) {
        this.color = color;
        this.resellPrice = resellPrice;
        this.retailPrice = retailPrice;
    }
    public abstract String getColor();
    public abstract int getResellPrice();
    public abstract int getRetailPrice();
    public void clean() {
        System.out.println("Swish Swish I'm clean!");
    }
}
```

```
class BataBoost700 extends Sepatu {
    public BataBoost700(String color, int resellPrice, int retailPrice) {
        super(color, resellPrice, retailPrice);
    }
    public String getColor() {
        return this.color;
    }
    public int getResellPrice() {
        return this.resellPrice;
    }
    public int getRetailPrice() {
        return this.retailPrice;
    }
}
```

Disini kita memanfaatkan OOP untuk memindahkan constructor nya ke Sepatu, sehingga di setiap jenis sepatu yang kita buat, kita tidak perlu menuliskan atribute berulang kali, tetapi **kita wajib untuk mengimplementasikan 3 method abstract tersebut.**

```
    public static void main(String[] args) {
        BataBoost700 bb700 = new BataBoost700("green", 100, 20);
        System.out.println("BataBoost700 resells for " + bb700.getResellPrice());
        BataBoost700 bb700v2 = new BataBoost700("pirate black", 200, 10);
        System.out.println("BataBoost700v2 resells for " + bb700v2.getResellPrice());
    }
```

Sekarang bila kita memasukan ini di main, maka yang akan ke print sama persis dengan OOP biasa:

BataBoost700 resells for 100

BataBoost700v2 resells for 200

Interfaces

Setelah mengerti abstract class, interface seharusnya sudah menjadi lebih mudah. Interface adalah mirip abstract class yang lebih tegas, karena interface **hanya boleh memiliki abstract methods. Concrete methods tidak boleh ada di interface.**

Dengan menggunakan contoh yang sama, maka akan menjadi seperti ini:

```
interface Sepatu {

    public abstract String getColor();

    public abstract int getResellPrice();

    public abstract int getRetailPrice();
}
```

Note constructor tidak ada disini, karena constructor merupakan concrete method, sehingga tidak boleh ada di interface.

Jadi sekarang misal kita ingin membuat sebuah class lain yang menggunakan ini, bila kita **tidak** mengimplement ketiga abstract method, maka akan error, jadi kita harus mengoverride nya seperti ini, **sama persis dengan abstract class, tapi dengan kata <u>implements</u>** :

```
class AirSwallow implements Sepatu {
   private String color;
   private int resellPrice;
   private int retailPrice;

   public AirSwallow(String color, int resellPrice, int retailPrice) {
       this.color = color;
       this.resellPrice = resellPrice;
       this.retailPrice = retailPrice;
   }

   public String getColor() {
       return this.color;
   }

   public int getResellPrice() {
       return this.resellPrice;
   }

   public int getRetailPrice() {
       return this.retailPrice;
   }
}
```

Penggunaannya pun sama seperti class biasa:

```
   public static void main(String[] args) {
       AirSwallow as = new AirSwallow("blue", 200, 10);
                   System.out.println("Air  Swallow  resells  for  "  +
as.getResellPrice());
   }
```

Output nya akan menjadi seperti:

Air Swallow resells for 200

Yang membuat interface powerful adalah **sebuah class bisa meng-implement banyak interface, tetapi hanya bisa mengextend 1 abstract class.** Dengan interface ini memungkinkan terjadinya *blackmagic* multiple inheritance.

Misalnya kita memiliki 2 interface seperti ini:

```
interface Sepatu {
   public abstract String getColor();
   public abstract int getResellPrice();
   public abstract int getRetailPrice();
}

interface BarangKenaPajak {
   public abstract void laporPajak();
}
```

Maka di class yang ingin kita buat, **kita bisa mengimplement lebih dari 1 interface, dipisahkan dengan koma**, tetapi kita harus mengoverride **semua** method yang ada di **semua** interface yang sudah kita implement

```
class AirSwallow implements Sepatu, BarangKenaPajak {
   private String color;
   private int resellPrice;
   private int retailPrice;
   public AirSwallow(String color, int resellPrice, int retailPrice) {
       this.color = color;
       this.resellPrice = resellPrice;
       this.retailPrice = retailPrice;
   }
// method dari interface Sepatu
   public String getColor() {
       return this.color;
   }
   public int getResellPrice() {
       return this.resellPrice;
   }
   public int getRetailPrice() {
       return this.retailPrice;
   }
// method dari interface BarangKenaPajak
   public void laporPajak() {
       System.out.println("Lapor");
   } }
```

Cheat Sheet Abstract Class & Interfaces

Abstract Class	Interface
abstract class Binatang{}	interface Binatang{}
Class Dog extends Binatang	Class Dog implements Binatang,..
Boleh memiliki abstract dan concrete methods (tidak harus memiliki keduanya)	Hanya boleh memiliki abstract methods saja
1 class hanya bisa mengextend 1 abstract class	1 class bisa mengimplement banyak interface
Boleh berisi constructor	Tidak boleh berisi constructor
Boleh memiliki public, private, protected attributes, functions, etc	Attribute pasti public static final sedangkan method pasti public abstract
Kapan menggunakan: Superclass dan subclass memiliki method yang harus beda tiap subclass (abstract) tetapi ada method yang sama untuk tiap subclass (concrete)	Kapan menggunakan: Ketika 1 class ingin meng inherit dari banyak class lainnya (multiple inheritance)
Tidak bisa di instantiate ~~Binatang cat = new Binatang()~~	Tidak bisa di instantiate ~~Binatang cat = new Binatang()~~

Soal Lab 07

Truk Donat

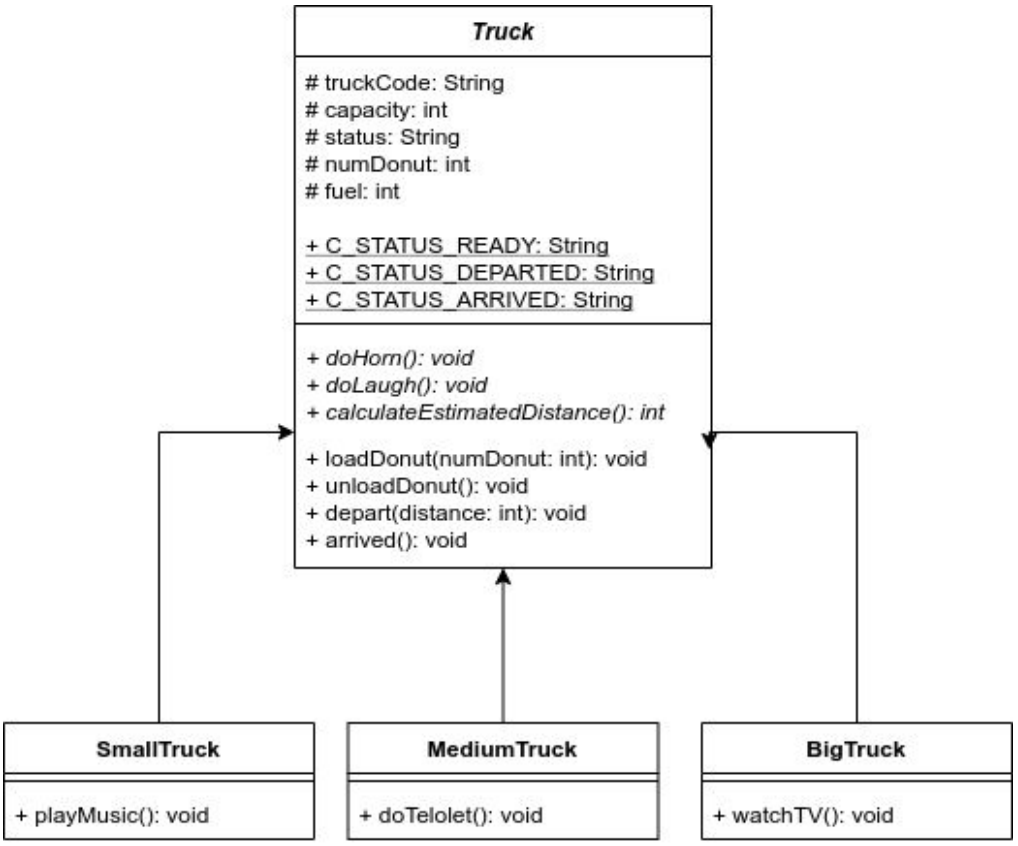
Source: spongebob.fandom.com

Namron ★ (baca: Namron Star) telah memiliki banyak toko donat yang tersebar di seluruh penjuru kota. Namun, akhir-akhir ini banyak pelanggan yang komplain karena rasa donatnya semakin tidak enak. Komplain tersebut berasal dari hampir seluruh toko miliknya, termasuk toko donat yang ia miliki di Bikini Bottom. Dari hasil wawancara Patrick Star mengatakan “Ya, donatnya semakin tidak enak, aku tidak akan membelinya lagi walaupun uangku tidak ada.”

Cuplikan wawancara bersama Patrick Star.

Setelah ditelusuri, ternyata kualitas donat Namron ★ berkurang karena pegawainya membuat donat secara asal-asalan. Untuk mengatasi hal tersebut, Namron ★ berencana untuk membuat sendiri donatnya, kemudian mendistribusikannya ke tiap tokonya menggunakan truk. Ada berbagai jenis truk yang akan dimiliki Namron ★, tetapi Namron ★ bingung untuk mendesain truknya. Bantulah dia.

Spesifikasi Program



- **Truck (Abstract Class)**
  - **loadDonut:**
    - Menambah donat ke dalam truk. Perhatikan kapasitas truk.
    - Method ini hanya dapat dipanggil ketika truk dalam keadaan **READY** dan belum berangkat.
    - Jika memenuhi keluarkan "`<newDonut> donat berhasil ditambah ke truk <truckCode>`".
    - Jika gagal keluarkan, "`Donat tidak dapat ditambah ke truk <truckCode>`". Tidak ada penambahan parsial, sehingga walaupun kapasitas masih ada, tetapi jumlah donat yang ingin ditambah lebih besar dari sisa kapasitas, maka tidak ada donat yang akan ditambah ke dalam truk.
  - **unloadDonut:**
    - Mengeluarkan donat dari dalam truk.
    - Method ini hanya dapat dipanggil ketika truk sudah dalam keadaan **ARRIVED**.
    - Jika berhasil, keluarkan "`<numDonut> donat berhasil dikeluarkan dari truk <truckCode>`".
    - Jika gagal, "`Truk <truckCode> tidak dapat mengeluarkan donat`".

    - **depart:**
        - Mengecek apakah truk dapat berangkat dengan jarak yang telah ditentukan.
        - Truk hanya dapat berangkat jika keadaan **READY**.
        - Jika berhasil, keluarkan "`Truk <truckCode> telah berangkat membawa <numDonut> donat`". Kemudian mengganti keadaan truk menjadi **DEPARTED**.
        - Jika gagal, keluarkan "`Truk <truckCode> yang membawa <numDonut> donat tidak dapat berangkat`".
    - **arrive:**
        - Truk sampai dan mengganti keadaan truk menjadi **ARRIVED**.
        - Truk hanya bisa sampai jika keadaan **DEPARTED**.
        - Jika berhasil, keluarkan "`Truck <truckCode> telah sampai`".
        - Jika gagal, keluarkan "`Truck <truckCode> sepertinya belum berangkat`"
    - **doHorn (Abstract Method):**
        - Melakukan klakson, disesuaikan dengan jenis truk.
    - **doLaugh (Abstract Method):**
        - Pengemudi tertawa, disesuaikan dengan jenis truk.
    - **calculateEstimatedDistance (Abstract Method):**
        - Menghitung jarak yang dapat ditempuh truk dari bahan bakar yang ada, disesuaikan dengan jenis truk.
- **SmallTruck**
    - **doHorn:**
        - Keluarkan "`Truk <truckCode> tin.. tin..`"
    - **doLaugh:**
        - Pengemudi yang mengendarai truk ini akan tertawa seperti Spongebob "HAHAHAHAHA". Keluarkan "`Pengemudi truk <truckCode> tertawa HAHAHAHAHA`".
    - **calculateEstimatedDistance:**
        - Satu satuan bahan bakar sama dengan 100 satuan jarak.
    - **playMusic:**
        - Keluarkan "`Pengemudi truk <truckCode> memutar lagu`".
- **MediumTruck**
    - **doHorn:**
        - Keluarkan "`Truk <truckCode> tet.. tet..`"
    - **doLaugh:**
        - Pengemudi yang mengendarai truk ini akan tertawa seperti Squidward "HEKHEKHEK". Keluarkan "`Pengemudi truk <truckCode> tertawa HEKHEKHEK`".
    - **calculateEstimatedDistance:**
        - Satu satuan bahan bakar sama dengan 75 satuan jarak.

    - **doTelolet:**
        - Keluarkan "`Pengemudi truk <truckCode> membunyikan telolet.. telolet..`"
- **BigTruck**
    - **doHorn:**
        - Keluarkan "`Truk <truckCode> tot.. tot..`"
    - **doLaugh:**
        - Pengemudi yang mengendarai truk ini akan tertawa seperti Mr. Krab "EIKEIKEIEKEIEKIEKEIK". Keluarkan "`Pengemudi truk <truckCode> tertawa EIKEIKEIEKEIEKIEKEIK`".
    - **calculateEstimatedDistance:**
        - Satu satuan bahan bakar sama dengan 50 satuan jarak.
    - **watchTV:**
        - Keluarkan "`Pengemudi truk <truckCode> menonton TV`".

Contoh

SampleOutput.txt

Komponen Penilaian

- 50% Implementasi Truck.
- 40% Implementasi SmallTruck, MediumTruck, dan BigTruck.
- 10% Dokumentasi dan kerapian kode.

Kumpulkan berkas .java yang telah di-zip dengan format penamaan seperti berikut.
`Lab07_[Kelas]_[KodeAsdos]_[NPM]_[NamaLengkap].zip`
Contoh:
`Lab07_A_SMA_1234567890_DekDepe.zip`