

Java OOP (Part 2)

Silver - Chapter 2 - Topic 4

**Selamat datang di Chapter 2 Topic 4
online course Back End Java dari
Binar Academy!**



Heyho! Siapa yang udah siap belajar hari ini? 😊

Pada topik sebelumnya kita udah kenalan sama Konsep Java. Berhubung topik sebelumnya kayaknya seru banget, kita jadi lanjut ke part 🤝

Kalau kamu masih inget sama dua pilar OOP dan penasaran dengan dua pilar lainnya, topik ini hadir untuk kamoe~

Selain itu, ada materi baru yang makin menambah wawasan kamu tentang **konsep dari Java**. Yuk, kita buka tirai selanjutnya ➡



Dari sesi ini, kita bakal bahas hal-hal berikut:

- Dua pilar OOP - Abstraction dan Inheritance
- Konsep Interface dan Abstract Class
- Keyword Static dan Final
- Pengantar Enum
- Pengantar Data Transfer Object (DTO)



The day is coming~

Ada empat pilar OOP yang terdiri dari:

- Encapsulation,
- Polymorphism,
- Abstraction, dan
- Inheritance.

Melanjutkan dari topik sebelumnya, kita bakal jelasin tentang pilar yang ketiga yaitu **Abstraction**.



Abstraction disini adalah sebuah aspek yang berfungsi untuk **menyembunyikan detail dari suatu aktivitas terhadap user**.

Detail aktivitas tersebut dianggap sebagai sesuatu yang nggak penting, jadi harus disembunyikan biar mempermudah kerjaan kamu.



Jadi, Abstraction itu apa sih?

Kamu pernah kan ditanya sama temen tentang kegiatan kamu hari ini:

“Kamu ngapain aja?”

Hmm.. kamu seneng buat ditanyain, tapi kamu nggak menceritakan semua kegiatan yang dilalui karena khawatir hal tersebut nggak penting dan dia juga nggak perlu tahu.

Pernah merasa kayak gitu? Yap, abstraction di Java juga mirip-mirip kayak kamu..

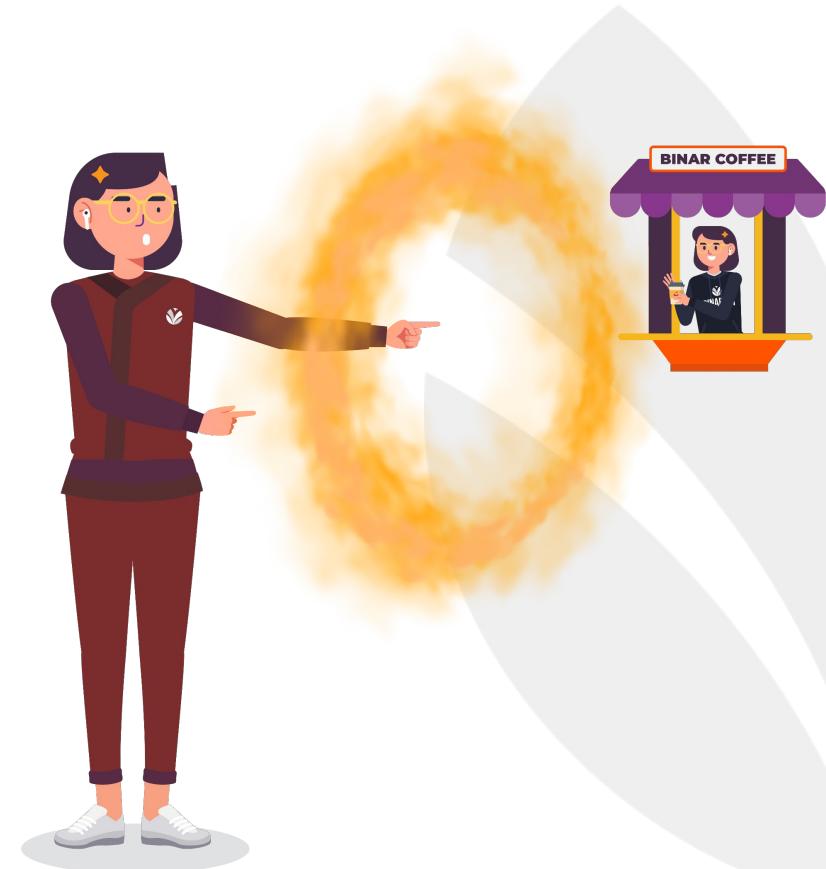


Ada contoh cerita lagi, nih.

Misalnya hari ini kamu disuruh kerja untuk membuat report. Dari kalimat instruksi terkait, nggak ada penjelasan tentang gimana report itu bisa dibuat.

Apakah dibuat dengan menggunakan personal computer? smartphone? ataupun tulis tangan? Apakah boleh ngerjain di cafe atau harus ke kantor?

Informasi yang disajikan cuma gitu aja yang intinya disuruh bikin report.



Dalam pemrograman OOP, **abstraction** dilakukan untuk membuat generalisasi dari suatu object.

Hal ini digunakan untuk mempermudah dalam memahami struktur kode yang kompleks dan mengurangi penulisan code yang berulang-ulang.

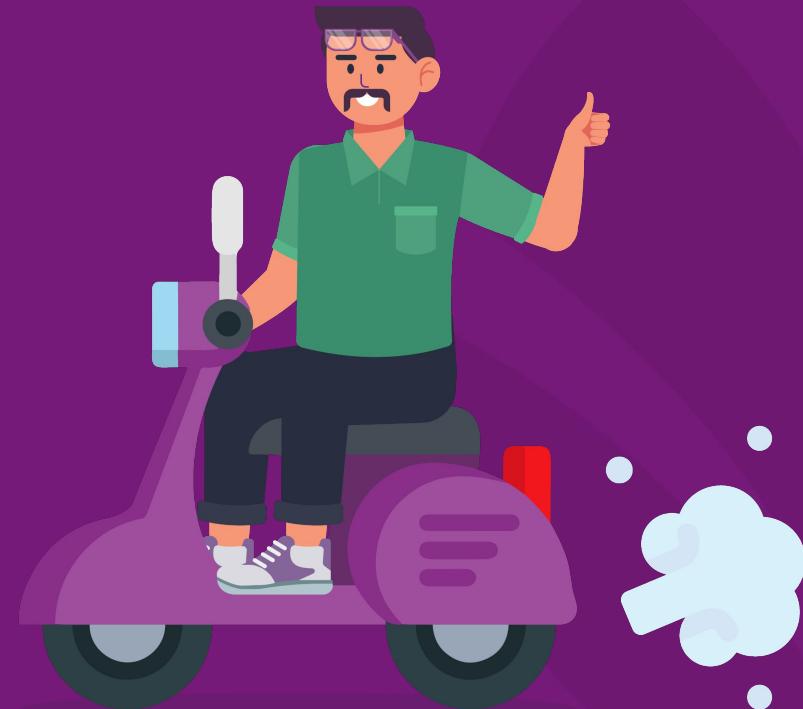
Untuk melakukan abstraction di Java, kita bisa pakai dua cara, yaitu membuat **abstract class**, dan membuat **interface**.



Java memiliki dua cara untuk membuat Abstraction, yaitu **Interface** dan **Abstract Class**.

Apa sih itu **Interface** dan **Abstract Class**?

Yep, pertanyaan kita sama. Which is bakal kita cari tahu setelah slide ini. Let's go!



Cara yang pertama adalah Abstract Class~

Abstract class bisa dibilang sebagai class yang punya keyword abstract. Abstract juga diperbolehkan untuk memiliki abstract method.

Abstract method? apakah itu?

Abstract method merupakan pendeklarasian method tanpa implementasi dari codenya. Pendeklarasian ini cuma berupa nama method, return type/void, dan parameternya.



Selain boleh punya satu abstract method atau lebih, abstract class juga boleh punya method non-abstract.

Eh tapi ada catatan besar, walaupun bisa non-abstract tapi abstract class nggak bisa di-instance.

Abstract class cuma bisa diinstance kalau abstract class udah punya subclass. Tentang subclass ini bakal dibahas di subtopic Inheritance, ya! Santai dulu kita~



Berikut adalah contoh abstract class Person!

Kalau kita perhatikan nih, pada class Person, ada satu method abstract yaitu work() yang nggak punya implementasi. Tapi, ada juga satu method lain yaitu changeName yang punya implementasi.

Jadi, abstract class diperbolehkan punya sebuah field dan constructor.

```
● ● ●  
public abstract class Person {  
  
    private String name;  
    private String gender;  
  
    public Person(){}
  
  
    public abstract void work();
  
  
    protected void changeName(String newName)
    {
        this.name = newName;
    }
}
```

Selain abstract class, abstraction juga bisa dibuat pake interface~

Beda sama abstract class, interface cuma berisi abstract method dan dan nggak punya field sama sekali.

Selain itu, interface nggak punya constructor dan nggak bisa di-instance.



Berikut adalah contoh dari interface!

Pada contoh di samping, ada dua method yaitu register dan getRegisteredPerson yang nggak punya implementasi.

Buat nulis implementasi dari abstract method di interface, **interface harus diimplement oleh suatu class.**

```
Public interface Registration {  
    void register(String  
name);  
    Person  
getRegisteredPerson(String  
name);  
}
```

Ibarat rumah yang pilarnya harus lengkap biar jadi kokoh. Kalau di Java, pilar terakhir untuk melengkapi OOP adalah **Inheritance**.

Langsung aja, kita berangkat~



Konsep Pewarisan dalam Java~

Inheritance atau pewarisan adalah konsep **mewariskan field dan method pada class turunannya**.

Tujuannya buat **reusability** atau penggunaan kembali pada code yang sudah dibuat.



Misalnya tuh kayak kamu mau bikin class baru tapi mirip sama class yang udah ada, punya field serta method yang sama juga.

Ibaratnya tuh, kenapa harus buat dari awal, kalau kamu bisa langsung mewariskan field dan method dari class yang udah ada? Iya, kan?!

Hanya data yang bisa
aku wariskan nak
bukan harta



Inheritance bisa membuat sebuah struktur atau hirarki pada class, lho!

Konsep pewarisan dalam inheritance dikenal dengan dua istilah, yaitu superclass dan subclass.

- **Superclass** adalah class lama atau class yang mewariskan fitur, atau bisa dikatakan sebagai class induk.
- **Subclass** merupakan class turunan atau yang mewarisi isi dari superclass, atau sebagai class anak.



Biar kamu makin paham, kita coba pakai contoh, ya~

Kamu tahu Mufasa dan Simba di The Lion King?

Kalau dikaitkan sama Inheritance, Mufasa sebagai ayah dari Simba adalah **superclass**. Sedangkan Simba sebagai anak, adalah **subclass**.



Sekarang kita coba bedah cerita The Lion King, ya!

Selain Mufasa dan Simba, ada lagi nih satu karakter yang ikonik yaitu Rafiki.

Penasihat Mufasa ini pernah bilang ke Simba, kalo jiwa Mufasa ada dalam diri Simba atau **Simba mewariskan sifat-sifat ayahnya** yang pemberani dan baik hati.

Nah, sifat yang diwariskan dari Mufasa ke Simba ini adalah contoh dari Inheritance.



Hubungan antara abstraction dengan inheritance~

Kayak yang udah dibahas di subtopic abstraction, kita tahu bahwa abstract class nggak bisa diinstance kecuali terdapat subclass dari abstract class tersebut.

Selain itu untuk menulis implementasi abstract method dari suatu interface, sebuah class yang mengimplementasi interface tersebut perlu dibuat dulu.

Hal ini sebenarnya berkaitan dengan konsep inheritance.



Penulisan implementasi dari sebuah abstract method dilakukan menggunakan **overriding**, yaitu **method yang namanya sama, return yang sama, dan parameter yang sama**.

Tapi, implementasi dari method tersebut bisa beda-beda. Method yang dibuat implementasinya harus pake annotation @Overriding.

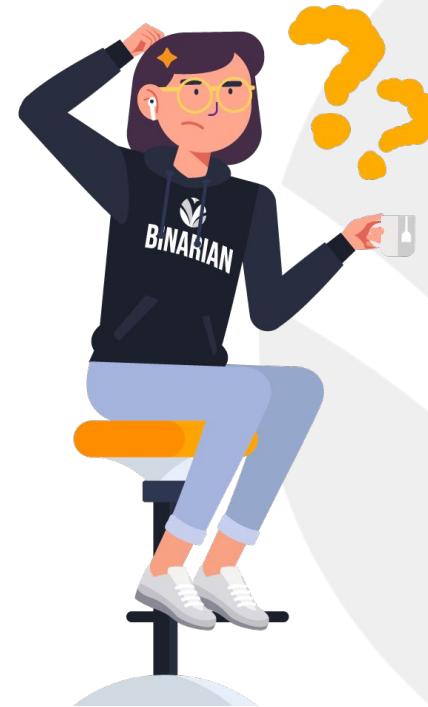


Kalau gitu, cara bikin superclass dan subclass itu gimana, dong?

Untuk membuat suatu subclass, subclass harus **meng-extend** sebuah class.

Pas subclass yang berupa non-abstract class meng-extend sebuah abstract class, berarti semua abstract method di abstract class harus ditulis implementasinya.

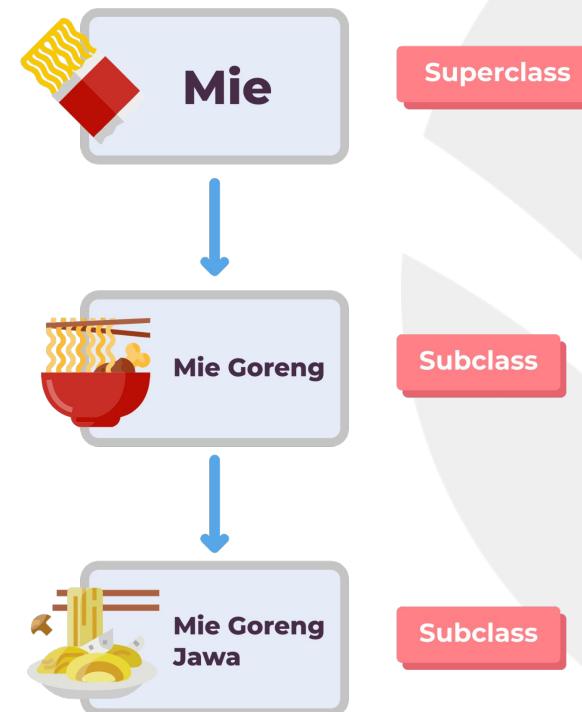
Dengan kata lain, abstract method di class abstract class di-override sama subclass yang merupakan non-abstract class.



**Masih ada lagi nih sob
lanjutannya~**

Di samping ini adalah contoh kalau sebuah subclass yang punya superclass, masih bisa punya subclass lagi. Jadi nggak terbatas hanya punya satu subclass aja.

Keren, ya~



Sekarang, kita coba lihat contoh berikut, ya!

Ini adalah contoh dari **superclass**.

Subclass dari person ada pada slide berikutnya.

```
@Getter  
@Setter  
@NoRequiredArgs  
public abstract class Person {  
  
    private String name;  
    private String gender;  
  
    public abstract void work();  
  
    protected void changeName(String newName)  
    {  
        this.name = newName;  
    }  
}
```

Class Employee merupakan subclass dari Abstract class person yang ditandai sama **keyword extends**.

Class Employee bisa mengakses method dan field pakai access modifier default dan protected, contohnya changeName.

Kalau Person bukan abstract class, subclass Employee nggak perlu melakukan overriding pada abstract method karena non-abstract class nggak punya abstract method.

```
public class Employee extends Person{
    private String EmployeeId;
    public Employee(){
        super();
    }
    @Override
    public void work(){
        System.out.println("Working");
    }
}
```

Terus, cara pake interface buat inheritance gimana?

Untuk melakukan inheritance menggunakan interface, keyword yang dipakai adalah **keyword implements**.

Implements digunakan untuk membuat implementasi dari abstract method yang ada di interface.



Selain implementasi, sebuah interface juga mewariskan abstract method -yang dimiliki interface lain- dengan keyword extends, tanpa harus menulis implementasinya.



Berikut adalah contoh dari Interface yang meng-extends interface lain!

Coba kita kulik gambar di samping. Diketahui Interface Sports memiliki 2 method.

Disini, Interface Football yang meng-extend interface Sports memiliki:

- 2 method yang dideklarasi di dalam interfacenya sendiri, dan
- 2 method yang diwarisi oleh interface Sports.



```
public interface Sports {  
    void setHomeTeam(String name);  
    void setVisitingTeam(String name);  
}  
  
public interface Football extends Sports  
{  
    void homeTeamScored(int points);  
    void visitingTeamScored(int points);  
}
```

Dari method yang udah dijabarin tadi, menurut kamu ada berapa total method yang dimiliki oleh interface Football?

Yap, betul! total dari method yang dimiliki oleh interface Football adalah 4 method.

```
● ● ●

public interface Sports {
    void setHomeTeam(String name);
    void setVisitingTeam(String name);
}

public interface Football extends Sports
{
    void homeTeamScored(int points);
    void visitingTeamScored(int points);
}
```

Berikut adalah contoh dari Interface yang diimplementasi oleh class.

Class FootballMatch harus menulis implementasi dari seluruh abstract method yang memiliki interface Football.

Jadi, Football Match melakukan method overriding sebanyak 4 method.

```
public interface Football extends Sports {  
    void homeTeamScored(int points);  
    void visitingTeamScored(int points);  
}  
  
Public class FootballMatch implements Football{  
    @Overriding  
    public void setHomeTeam(String name){//some  
implementation};  
    @Overriding  
    public void setVisitingTeam(String name )  
{ //some implementation};  
    @Overriding  
    public void homeTeamScored(int points ){//some  
implementation};  
    @Overriding  
    public void visitingTeamScored(int points )  
{ //some implementation};
```

Lanjuttt

Kalau tadi kita udah kenalan sama keyword implements, selanjutnya kita bakal bahas keyword **Static and Final**.

Are you ready?





Keyword Static dan Final itu apa?

Kedua keyword ini dipakai Java, khususnya di class, field ataupun method.

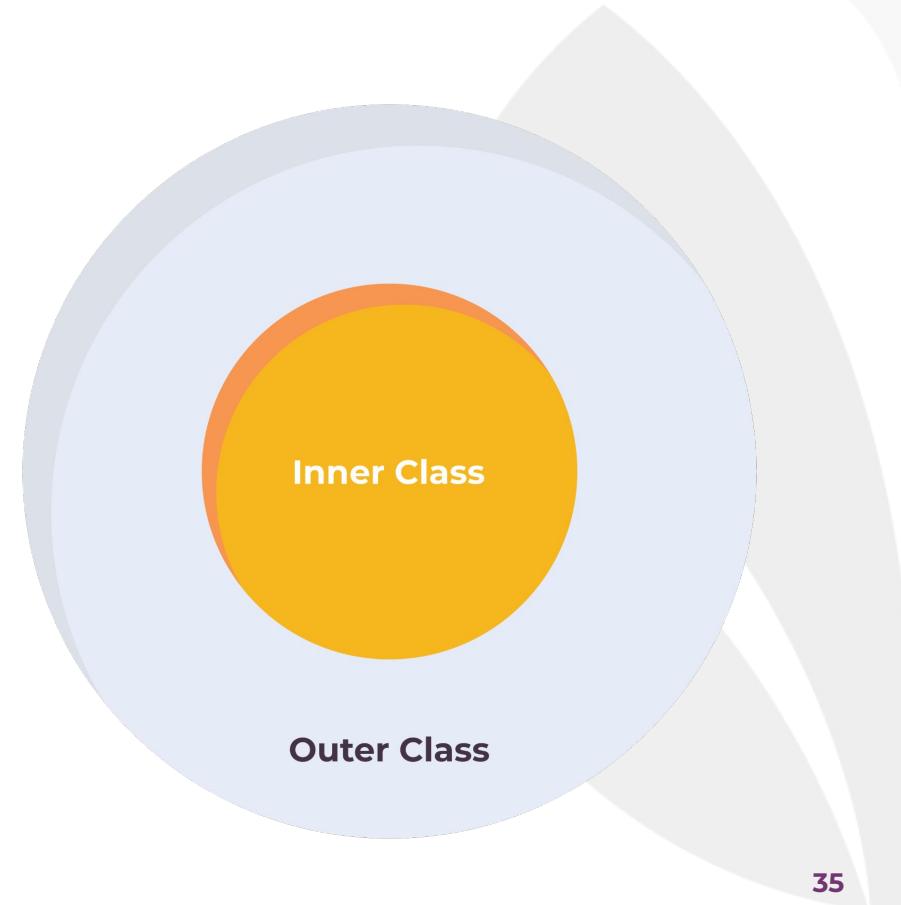
Kombinasi dari keyword static dan final pada class, baik field ataupun method, punya fungsi yang beda-beda.

Apa aja ya fungsinya?

Penggunaan keyword static di class cuma berlaku di sebuah nested class.

Kamu pernah dengar istilah itu?

Yep, **nested class adalah class yang berada di dalam suatu class**. Additionally, pada nested class ini, ada dua istilah yaitu inner class dan outer class. Inner class adalah class yang berada di dalam outer class.



Penggunaan **keyword static cuma berlaku pada inner class aja.**

Penggunaan static keyword pada inner class akan membuat proses instance inner class menjadi bisa dilakukan tanpa harus meng-instance outer classnya terlebih dahulu.

Jadi lebih sederhana, kan?



Berikut adalah contoh penggunaan static class pada nested class!

Ada dua inner class yang ada. Hayo, kamu bisa tebak nggak kira-kira ada apa aja?

```
public class Outer {
    public Outer(){}
    static class StatNestedDemo {
        public void myMethod() {
            System.out.println("This is a static class");
        }
    }
    class NonStatNestedDemo {
        public void myMethod() {
            System.out.println("This is not a static
class");
        }
    }
    //di dalam class lain
    public static void main(String args[]) {
        Outer.StatNestedDemo nested = new
Outer.StatNestedDemo();
        nested.myMethod();
        Outer outer = new Outer();
        Outer.NonStatNestedDemo nonStatNested =
outer.new NonStatNestedDemo();
        nonStatNested.myMethod();
    }
}
```

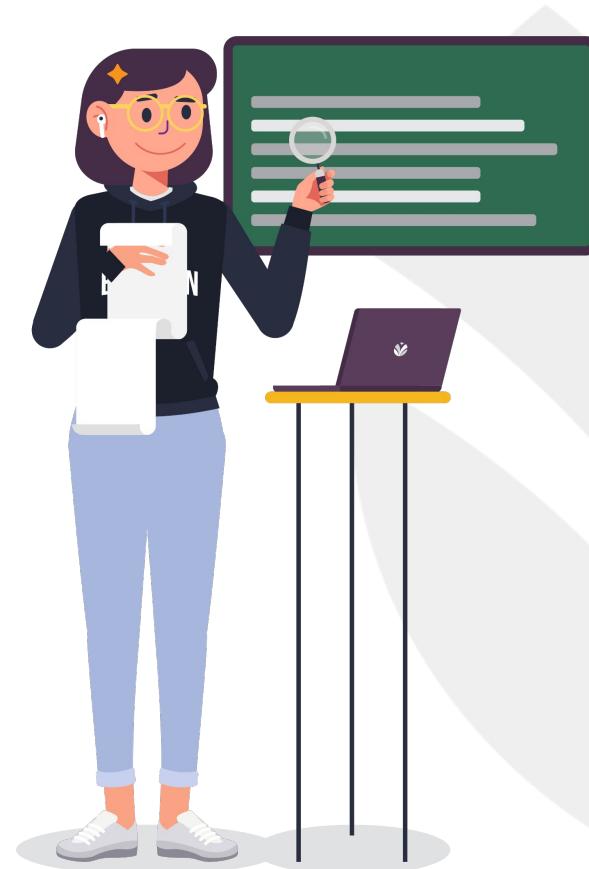
Pada slide sebelumnya, ada contoh penggunaan static class pada nested class.

Ada dua inner class yang ada yaitu:

- **StatNestedDemo** yang merupakan static class
- **NonStatNestedDemo** yang merupakan non static class

Pas StatNestedDemo di-instance, maka class Outer yang merupakan outer class nggak perlu di-instance.

Sedangkan pada NonStatNestedDemo, class Outer harus di-instance terlebih dahulu.



Penggunaan keyword final pada class menyebabkan **suatu class nggak bisa memiliki subclass**.

Artinya, class lain nggak bisa meng-extend class yang punya keyword final.

Sebuah abstract class jangan dibuat menjadi final class, soalnya abstract class butuh subclass supaya bisa di-instance.



Berikut contoh dari final class!

```
public final class Employee extends Person{  
    //class content  
}
```

Class Employee yang merupakan final class diperbolehkan untuk memiliki superclass, tapi class Employee dilarang untuk final class.



Jadi, fungsi Static Method itu buat apa?

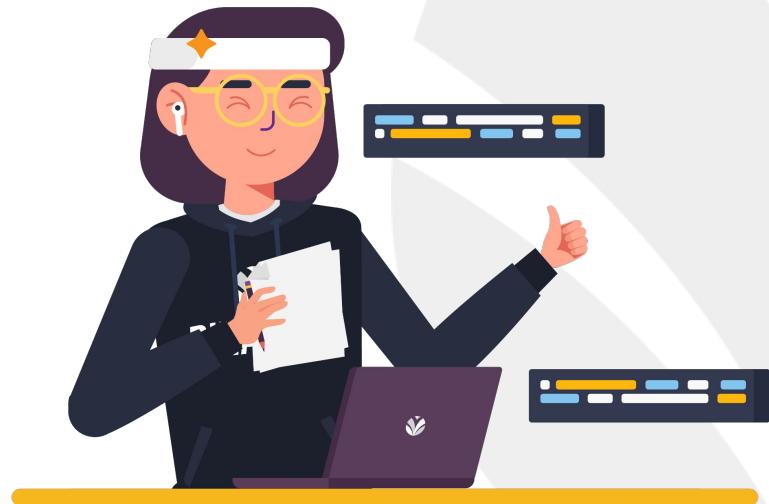
Keyword static pada sebuah method berfungsi supaya method bisa digunakan tanpa harus meng-instance class yang punya method tersebut.

Biasanya **dipakai untuk keperluan class utility**, yaitu class yang methodnya punya kerjaan sederhana dan sering di-invoke/dipanggil oleh class lain.



Nah berikut ini adalah contoh dari Class Utility:

```
public class Utils {  
    public static String  
    serializeMessage(Object data) {  
        if ((byte[]) data == null)  
            return "";  
        return new String((byte[]) data);  
    }  
}
```



Buat pake method serializeMessage, kita cuma bisa manggil methodnya tanpa harus meng-instance class utils.

Utils.serializeMessage(anyObject);



Kalau yang Final Method, gimana?

Gini, gini.

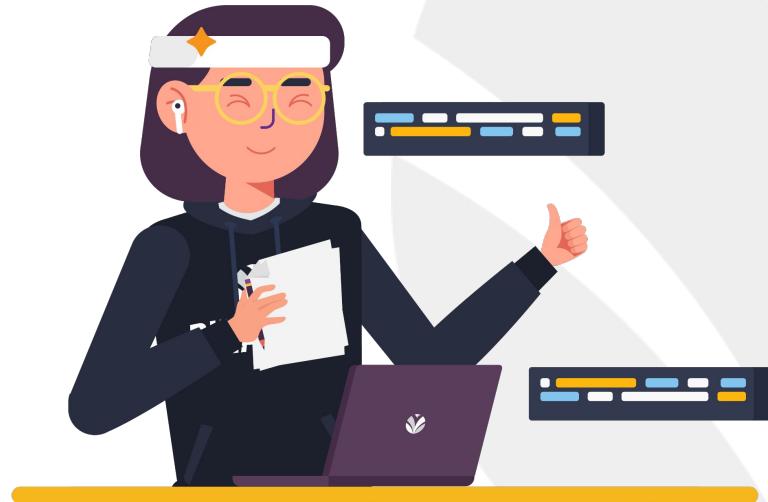
Penggunaan keyword `final` pada suatu method menyebabkan method -yang pake `final` keyword- menjadi method yang anti dengan implementasi baru ke subclass-nya.

Artinya, **final method memfinalisasi implementasi dari method, sehingga nggak bisa di-override.**

Berikut adalah contoh penggunaan final method!

```
public final void myMethod() {  
    System.out.println("This is not  
a static class");  
}
```

Artinya kalau class yang punya myMethod di-extend sama subclass, berarti subclass nggak bisa meng-override method tersebut.



Sekarang kita bahas konsep dari Static pada Field, yuk!

Keyword static pada sebuah field **menyebabkan dua object yang berbeda -tapi berasal dari class yang sama-** bisa akses value dari field antara satu dengan lainnya.



Kalau value dari static field di object A berubah, maka value dari static field di object lain juga ikut berubah.

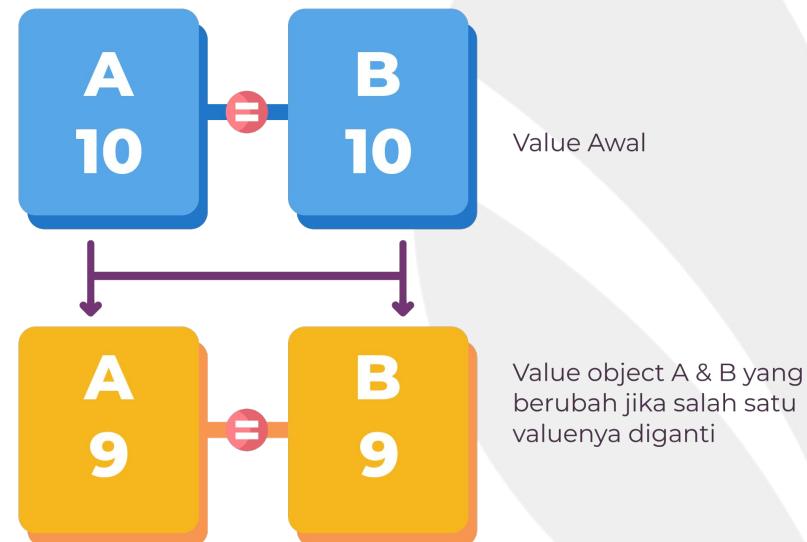
Konsep static pada field ini mirip kayak quotes anak zaman sekarang yang “kalo kamu berubah, aku juga berubah~”.



Biar lebih ajib, pakai contoh deh~

Kita punya dua object, yaitu object A dan object B yang punya value masing-masing bernilai 10.

Misalnya value object A kita ubah jadi 9, value object B bakal berubah jadi 9. Begitupun sebaliknya, kalo kita ganti value object B jadi 8, nantinya value object A ikut jadi 8.



Karena yang Static pada Field udah ada gambaran, yang ini adalah konsep dari Final pada Field~

Keyword final di sebuah field menyebabkan **value pada suatu field nggak bisa dimodifikasi lagi.**

Dari situ, kamu cuma bisa meng-assign value pada pertama kali aja.



Pertama kali gimana, tuh?

Kamu nggak bisa buat modifikasi lagi value tersebut karena value yang diassign pertama kali akan menjadi value yang final.

Value-nya bisa berbeda-beda pada setiap object.



Contohnya, kita punya object A yang punya value 10. Terus kita ganti value-nya jadi 9.

Walaupun diganti Value-nya akan tetap bernilai 10 karena udah nggak bisa dimodifikasi.

Kayak kita yang kalau udah dibohongin sekali, pasti udah nggak bakal percaya lagi~



Sekarang kita bahas kombinasi Static Final pada Field

Penggunaan static final pada suatu field bakal membuat field tersebut menjadi sebuah constant.

Constant adalah nilai dari suatu field yang bersifat tetap dan nggak bisa dimodifikasi. Permanen deh pokoknya~



Pendeklarasian sebuah constant di Java biasanya pake **keyword static final** dan penamaannya pake **huruf kapital**.

Di proses compilation, semua constant bakal di-compile jadi lebih awal.



Berikut contoh penggunaan static final pada field untuk membuat suatu constant:

```
● ● ●  
public static final String APP_CONSTANT  
= "anyApp";
```

Contoh di atas adalah pendeklarasian constant APP_CONSTANT yang isinya String "anyApp".

Isi dari string tersebut nggak bisa diubah lagi. Jadi, harap ingat baik-baik ya, guys!



Sipp, deh! Kayaknya udah bisa ngebedain static dan final, nih.

Setelah ngobrolin tentang doi yang berubah, abis ini kita belajar tentang **Enum**.

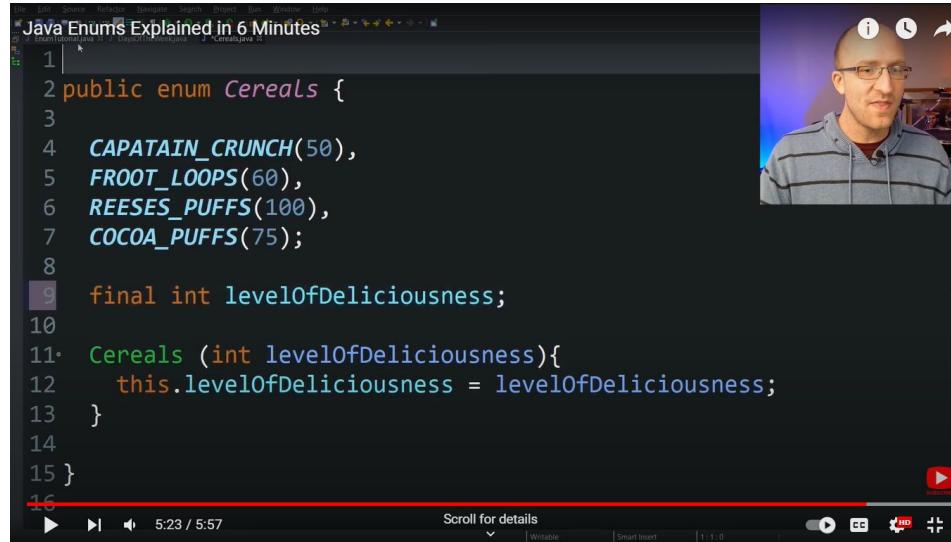
Bukan~bukan~ Bukan typo, ya. **Enum** ini ada kaitannya nih sama class.



Enum merupakan suatu class khusus untuk menyimpan sekelompok constant.

Biar makin paham, simak contoh penggunaan enum di samping ini👉

```
● ● ●  
class EnumExample{  
    enum Season{  
        WINTER(5), SPRING(10), SUMMER(15),  
        FALL(20);  
  
        private int value; //merupakan constant  
        berupa angka  
        private Season(int value){  
            this.value=value;  
        }  
    }  
    public static void main(String args[]){  
        for (Season s : Season.values())  
            System.out.println(s+" "+s.value);  
    }  
}
```



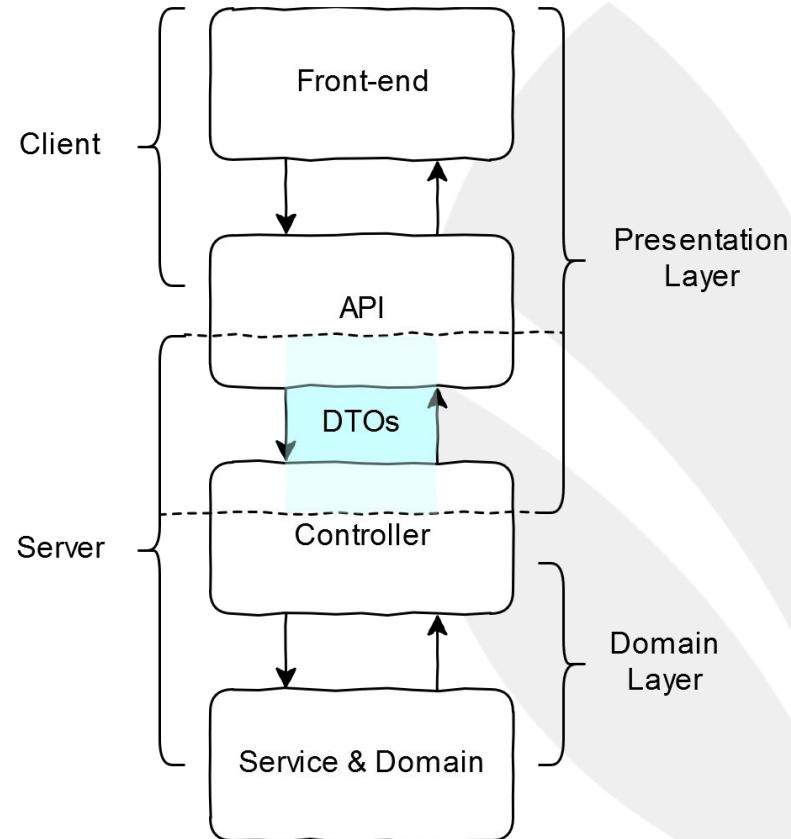
Kalau kamu mau eksplor Enum lebih lanjut, di bawah ini ada referensi video yang berisikan tentang gimana cara membuat Enum, gimana cara menggunakan, dan dalam situasi apa kamu bisa menggunakan Enum. Yuk, kita tonton dulu videonya

[Java Enums Explained in 6 Minutes](#)

Wuiw! Kita masuk ke materi terakhir kita, yaitu **DTO** atau **Data Transfer Object**.

Kira-kira transfer kayak gimana ya yang dimaksud?



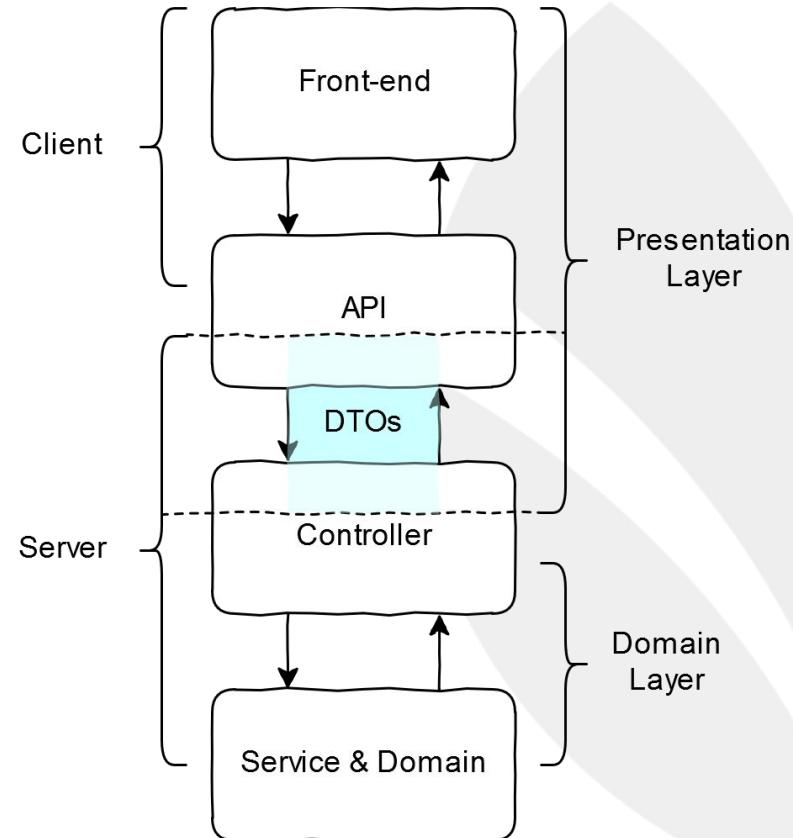


DTO adalah kependekan dari Data Transfer Object yang artinya **object yang dipakai untuk meng-encapsulate data yang ada supaya dapat berkomunikasi dengan class lain.**

Pada Java, udah jadi hal yang biasa banget untuk membuat proses pemecahan jadi berbagai macam object. Proses tersebut disebut parsing object.

Parsing object adalah sebuah proses pemindahan data dari bentuk suatu object ke object lain.

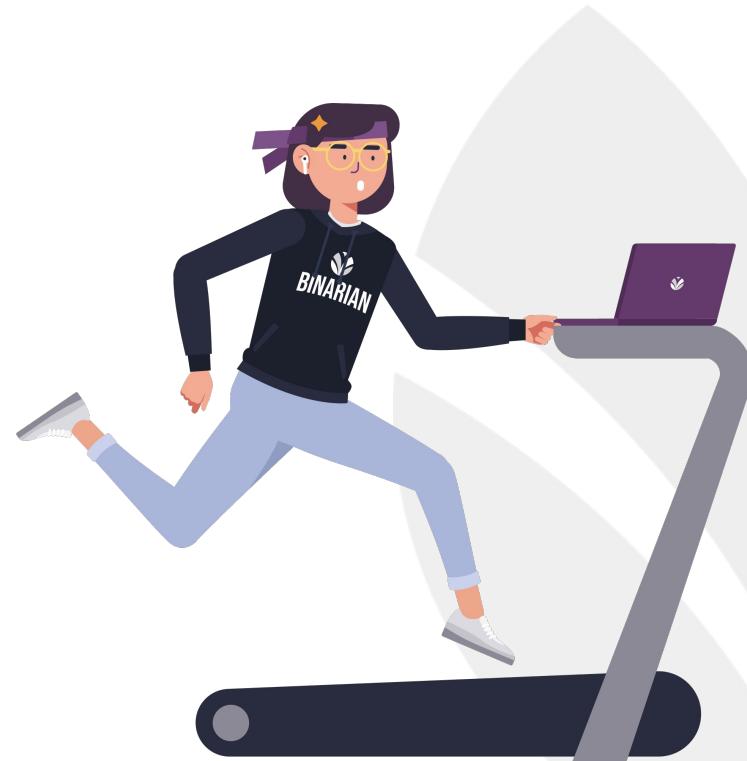
Dalam melakukan proses parsing, tetep pakai kaidah OOP yang benar. Yaitu pakai encapsulation, dengan memanfaatkan method getter dan setter.



Wah, ada misi lagi nih!

Kalau di pertemuan sebelumnya kamu sudah latihan mengimplementasikan dua pilar OOP, pada pertemuan kali ini kamu diminta untuk **memuat implementasi 4 pilar OOP (encapsulation, polymorphism, inheritance dan abstraction) dengan konsep Java**.

Sama seperti kemarin, diskusikan jawabannya bersama teman sekelas dan facilitator ya!



Wah asyik, kamu udah belajar semua 4 pilar OOP di chapter ini 🎉

Coba deh kamu renungkan, dengan pilar OOP ini kamu bisa buat atau merancang apa aja sih? Coba bisikan jawaban kamu ke Sabrina dong~



Nah, selesai sudah pembahasan kita pada topik 4 ini 😊

Selanjutnya, kita bakal siap-siap untuk move on ke topik baru yaitu:

✨Java Collections ✨

Sampai jumpa pada topik selanjutnya~

