

VERSION 1.0

17 Maret 2025



# PEMROGRAMAN BERORIENTASI OBJEK

MODUL 4 – PACKAGE, POLYMORPHISM, OVERLOADING, ABSTRACTION,  
INTERFACE

DISUSUN OLEH:  
WIRA YUDHA AJI PRATAMA  
KEN ARYO BIMANTORO

DIAUDIT OLEH:  
Ir. Galih Wasis Wicaksono, S.Kom, M.Cs.

PRESENTED BY: TIM LAB. IT  
UNIVERSITAS MUHAMMADIYAH MALANG

## PENDAHULUAN

### TUJUAN

1. Mahasiswa memahami konsep package, polymorphism, method overloading, interface, dan abstraction dalam Java.
2. Mahasiswa memahami bagaimana konsep-konsep tersebut digunakan untuk meningkatkan fleksibilitas dan efisiensi dalam pengembangan perangkat lunak berbasis OOP.
3. Mahasiswa dapat mengorganisasi kode dengan menggunakan package

### TARGET MODUL

1. Mahasiswa dapat membuat program yang menerapkan package, polymorphism, overloading, interface, dan abstraction dalam Java.
2. Mahasiswa dapat mengimplementasikan method overloading dan polymorphism untuk meningkatkan modularitas kode.

### PERSIAPAN

1. Device (Laptop/PC)
2. IDE (IntelliJ)

### KEYWORDS

Package, Polymorphism, Method Overloading, Interface, Abstraction

### TABLE OF CONTENTS

<b>PENDAHULUAN</b>	1
TUJUAN	1
TARGET MODUL	1
PERSIAPAN	1
KEYWORDS	1
TABLE OF CONTENTS	1
<b>PACKAGE</b>	3
TEORI	3
MATERI	3
Built-in Package	3
User-defined package (package buatan kita sendiri)	3
<b>POLYMORPHISM</b>	7
TEORI	7

<b>MATERI</b>	<b>8</b>
Ad Hoc Polymorphism (overloading method atau static polymorphism)	8
Dynamic Polymorphism (overriding method)	12
Object Casting	14
Heterogeneous Collection	17
<b>Abstraction</b>	<b>18</b>
Abstract Class	18
Abstract Method	21
Contoh Implementasi Abstraction	22
TIPS	25
<b>Interface</b>	<b>25</b>
Is-a relation & Operator instanceof	31
Has-a relation	32
Relasi lain	33
TIPS	33
<b>PRAKTEK</b>	<b>34</b>
<b>CODELAB &amp; TUGAS</b>	<b>40</b>
CODELAB	40
TUGAS	41
<b>PENILAIAN</b>	<b>45</b>
RUBRIK PENILAIAN	45
SKALA PENILAIAN	46
<b>SUMMARY AKHIR MODUL</b>	<b>47</b>

## PACKAGE

### TEORI

**Package** adalah sebuah **metode** untuk **mengelompokkan class** dengan tujuan **menghindari konflik nama class (jika ada yang bernama sama)** dan memudahkan pengelolaan kode program, terutama pada aplikasi yang besar.

Dalam implementasinya, konsep ini serupa dengan pembuatan folder saat menyimpan sebuah file. Meskipun setiap folder mungkin memiliki file dengan nama yang sama, namun karena disimpan dalam folder yang berbeda, hal tersebut tidak menjadi masalah.

Hal yang sama juga berlaku di dalam package Java, di mana kita dapat membuat nama class yang identik selama berada di dalam package yang berbeda. Package dalam bahasa pemrograman Java dibagi menjadi dua jenis:

1. Built-in package – package yang sudah disediakan oleh Java.
2. User-defined package – package yang kita definisikan sendiri.

### MATERI

#### Built-in Package

Java memiliki cukup banyak package bawaan dan beberapa yang sudah pernah kita pakai. Salah satu darinya adalah **java.util** yang berisikan **Scanner** class untuk proses input user. Untuk daftar package yang ada di java bisa klik [disini](#).

Untuk menggunakan package, tambah perintah **import** sebelum nama package di awal kode program, seperti **import java.util.Scanner**. Berikut adalah contoh proses import di dalam kode:

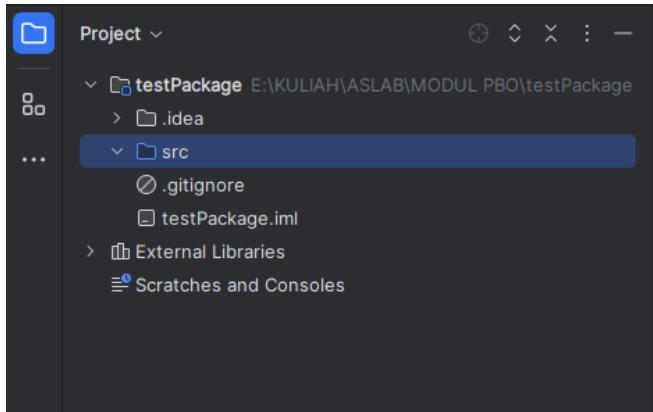
```
import java.util.Scanner;
```

Kode di atas digunakan untuk **mengimport Scanner class** yang ada di package **java.util**.

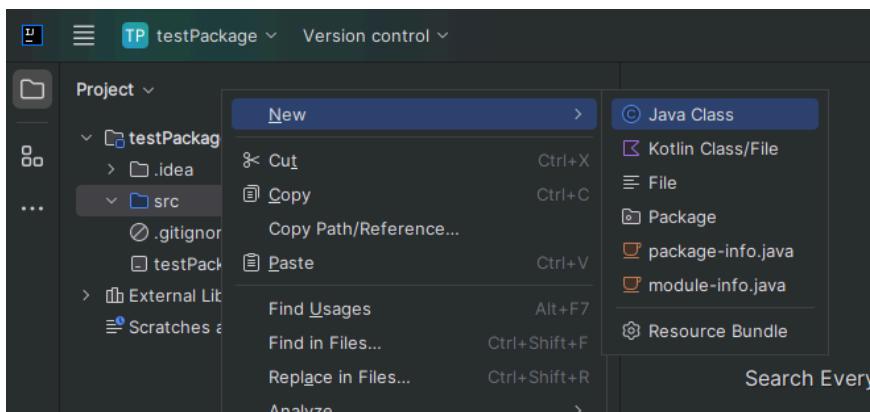
#### User-defined package (package buatan kita sendiri)

Sesuai dengan namanya package ini akan kita buat sendiri. Mari kita coba untuk membuat package sendiri dengan intelliJ IDEA.

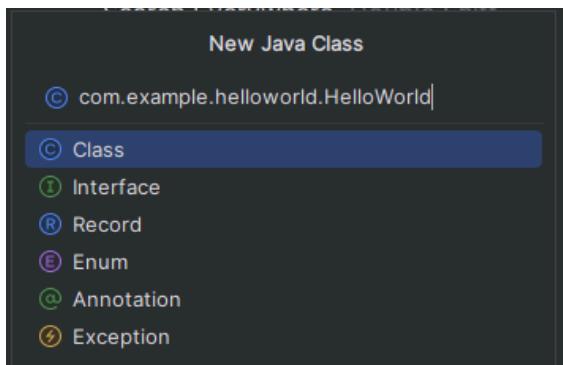
Silahkan buka intelliJ IDEA dan buat proyek baru nama dan location directory bebas sesuai dengan kebutuhan kalian, setelah itu klik create, ketika kita lihat pada tab bagian kiri, kita akan melihat seperti ini:



Di jendela **Project**, klik kanan pada folder **src**, pilih **New** (atau tekan **Alt+Insert**), lalu pilih **Java Class**.

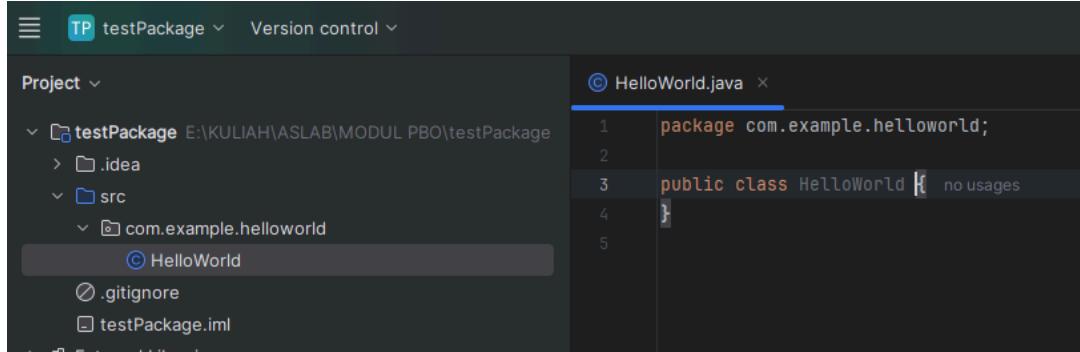


Di kolom **Name**, ketik **com.example.helloworld.HelloWorld** dan klik **OK/Enter**.

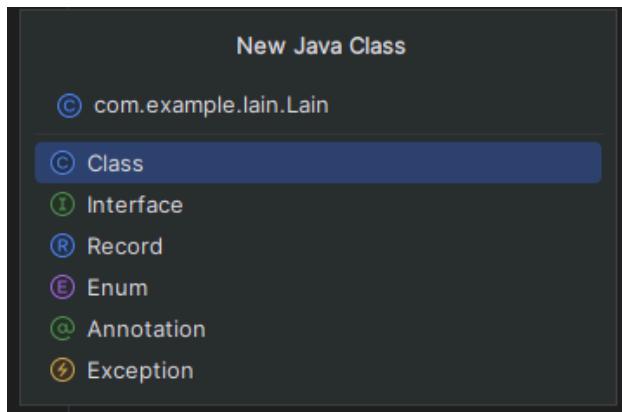


IntelliJ IDEA akan membuat package **com.example.helloworld** dan kelas **HelloWorld**.

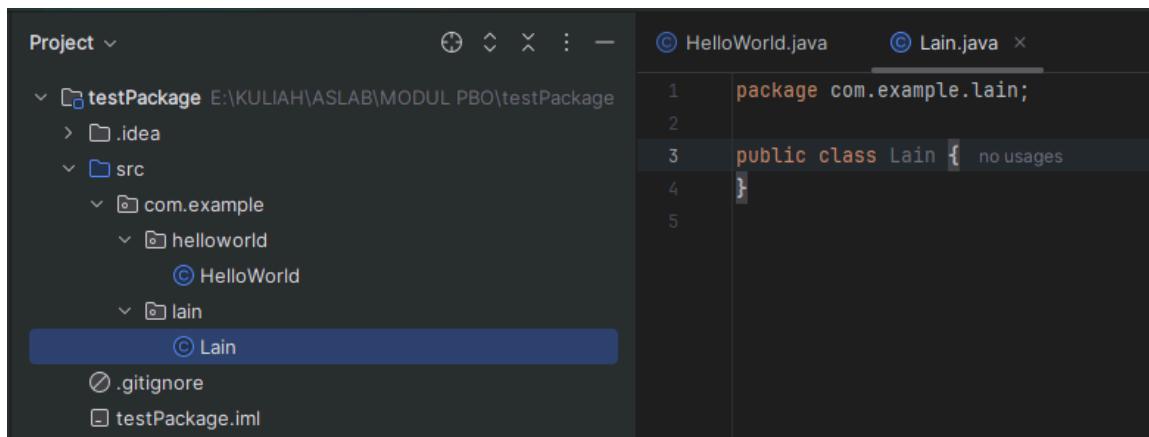
Maka di bagian kiri akan muncul package baru seperti ini:



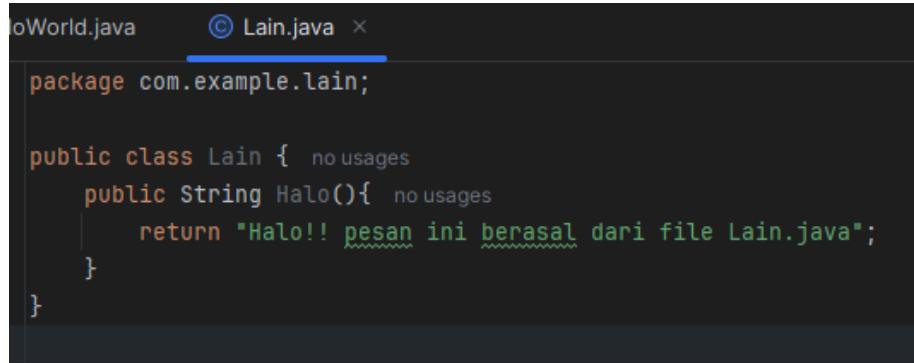
Untuk menambah file baru di dalam package baru juga, caranya sama seperti step sebelumnya ketika membuat package **helloworld**. Klik kanan pada folder **src**, pilih **New** lalu pilih **Java Class** dan masukkan nama file yang diinginkan.



Maka akan terbentuk sebuah file baru seperti ini:



Untuk cara pakai class “**Lain**” yang sudah kita buat di dalam package **com.example.lain**, kita coba untuk membuat sebuah method sederhana di dalam class tersebut. Contohnya kita buat sebuah method **halo()** dengan return type String yang berisi seperti ini:

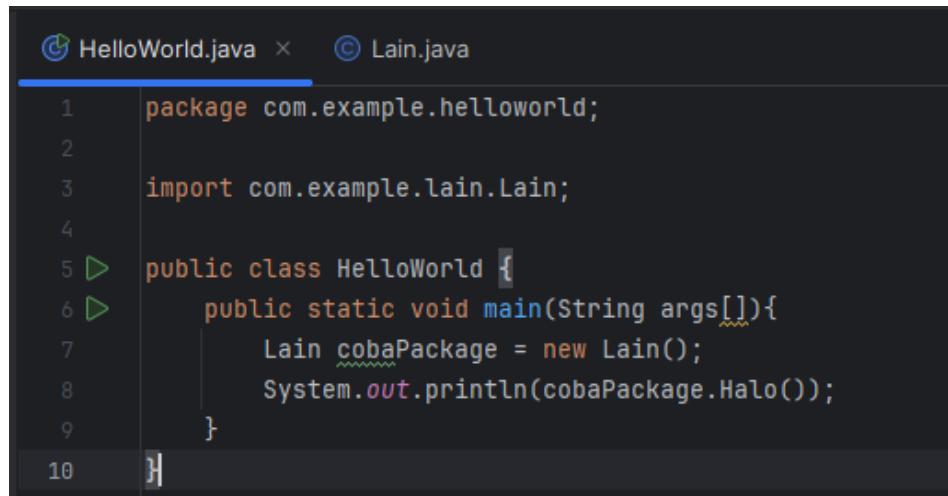


```
IoWorld.java    © Lain.java ×

package com.example.lain;

public class Lain { no usages
    public String Halo(){ no usages
        return "Halo!! pesan ini berasal dari file Lain.java";
    }
}
```

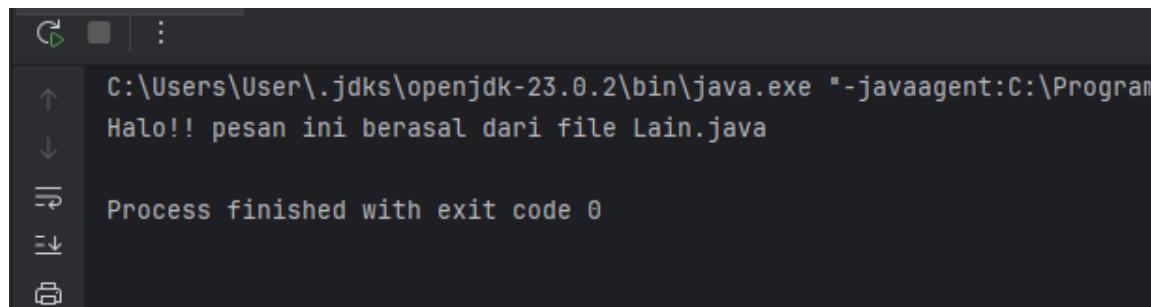
Untuk cara memakainya harus **import** terlebih dahulu class “**Lain**” di dalam class **HelloWorld**, lalu coba panggil method **Halo()** yang berasal dari kelas “**Lain**” seperti ini:



```
© HelloWorld.java ×  © Lain.java

1 package com.example.helloworld;
2
3 import com.example.lain.Lain;
4
5 public class HelloWorld {
6     public static void main(String args[]){
7         Lain cobaPackage = new Lain();
8         System.out.println(cobaPackage.Halo());
9     }
10 }
```

Ketika kita coba untuk run program-nya maka output kode akan seperti ini:



```
C:\Users\User\.jdks\openjdk-23.0.2\bin\java.exe "-javaagent:C:\Program
Halo!! pesan ini berasal dari file Lain.java

Process finished with exit code 0
```

Catatan tambahan untuk keywords **import**, ketika kita hanya ingin import 1 class saja contoh class **Scanner** maka kita bisa langsung menulis seperti ini **import java.util.Scanner;**. Sedangkan jika kita ingin mengimport banyak class maka harus menggunakan bintang “\*” untuk mewakili semua class. Contoh kita ingin import semua class yang terdapat pada package **java.util**, maka kita cukup ketikkan code seperti ini **import java.util.\*;**

## POLYMORPHISM

### TEORI

# Polymorphism



Polymorphism secara bahasa memiliki arti “**banyak bentuk**” atau “**bermacam-macam**”. Dalam konsep pemrograman, polymorphism sebuah teknik menggunakan fungsi atau atribut tertentu dari suatu parent class untuk diimplementasikan oleh child class baik secara default ataupun dimodifikasi sesuai dengan kebutuhan pada masing-masing class.

Polymorphism terjadi ketika kita **memiliki banyak class yang memiliki relasi satu sama lain menggunakan inheritance**. Jika sebelumnya kita sudah membuat sebuah class Hero yang memiliki child class (extends) yaitu Superman, Deadpool, dan Spiderman, dalam artinya class Hero memiliki banyak bentuk yaitu bisa menjadi superman, deadpool, superman atau hero yang lain. Contoh lain ialah sebuah kata “hewan”, lalu muncul di benak kita yaitu kucing, anjing, burung, buaya, dan lain-lain. Hal ini mengartikan **banyak bentuk** yaitu hewan bisa berupa kucing, bisa juga berupa anjing, bisa berupa burung ataupun buaya.

Contoh sederhana dari polymorphism adalah bagaimana seekor hewan bersuara. Kita tahu bahwa setiap hewan pasti bersuara, baik secara jelas ataupun tidak, namun pada intinya hewan pasti bersuara. Jika kita umpankan bahwa hewan bersuara sebagai method, dan method ini akan kita implementasikan pada semua hewan. Kucing jika mengimplementasikan method ini maka akan bersuara “Meow”, anjing mengimplementasikan method ini akan bersuara “Gug gug”, dan sapi mengimplementasikan method ini akan bersuara “Mooo”. Perbedaan dalam implementasi inilah yang menjadi salah satu contoh penerapan Polymorphism.

### MATERI

Terdapat beberapa tipe **Polymorphism** yaitu:

#### **Ad Hoc Polymorphism (overloading method atau static polymorphism)**

Ad Hoc Polymorphism, yang lebih dikenal dengan **method overloading**, adalah salah satu bentuk polimorfisme di mana **sebuah kelas memiliki beberapa metode dengan nama yang sama tetapi memiliki parameter yang berbeda**. Overloading method terjadi pada **compile time**, sehingga sering disebut sebagai static polymorphism.

Overloading Method memiliki beberapa karakteristik, diantaranya adalah :

- Metode memiliki **nama yang sama** tetapi berbeda dalam:
  - Jumlah parameter
  - Tipe parameter
  - Urutan parameter
- Overloading tidak bergantung pada **tipe kembalian (return type)**
- Diproses pada **waktu kompilasi** oleh compiler

Contoh pada class Hero sebelumnya kita ubah menjadi seperti ini:

```
public class Hero {  
    private String name;  
    public int umur;
```

```
public Hero(String name, int umur){  
    this.name = name;  
    this.umur = umur;  
}  
  
public Hero(String name){  
    this.name = name;  
}  
  
public Hero(){  
    // kosong  
}  
  
public void melindungi(){  
    System.out.println(name + " melindungi masyarakat");  
}  
  
public void setName(String name){  
    this.name = name;  
}  
  
public String getName(){  
    return name;  
}  
}
```

Terlihat pada kode di atas terdapat **3 constructor** dengan nama yang sama tetapi terdapat perbedaan **jumlah parameter dan tipenya**. Dengan adanya **overloading method** seperti di atas maka kita bisa membuat instance object dari class Hero seperti ini tanpa mengalami error:

```

src > Main.java > Main > main(String[])
1 public class Main {
2     Run | Debug
3     public static void main(String[] args) {
4         //membuat object dengan 2 argumen
5         Hero hero = new | 
6     }
    ⌄ Hero()
    ⌄ Hero(String name)
    ⌄ Hero(String name, int umur)
    ⌄ Main()

```

```

Main.java 3 • Hero.java
src > Main.java > Main
1 public class Main {
2     Run | Debug
3     public static void main(String[] args) {
4         //membuat object dengan 2 argumen
5         Hero hero = new Hero();
6         Hero hero2 = new Hero(name:"Ken");
7         Hero hero3 = new Hero(name:"Aryo", umur:22);
8     }

```

Terlihat pada kode di atas kita membuat object dengan 3 cara yaitu tanpa parameter, dengan 1 parameter, dan yang terakhir adalah dengan 2 parameter. Atau dengan contoh lain ialah kita coba ubah isi class Superman menjadi seperti ini:

```

public class Superman extends Hero{

    public Superman(String name, int umur) {
        super(name, umur);
    }
}

```

```

public void attack(){
    System.out.println("Superman menyerang musuh");
}

public void attack(String enemyName){
    System.out.println("Superman menyerang " + enemyName);
}

public void attack(String enemyName, int jumlah){
    System.out.println("Superman menyerang " + enemyName + " sebanyak " + jumlah);
}

public void terbang(){
    System.out.println(getName() + " terbang...");
}

@Override
public void melindungi(){
    System.out.println(getName() + " melindungi masyarakat bumi dari serangan monster");
}
}

```

Pada kode di atas kita membuat method **attack()** sebanyak 3 dengan jumlah parameter yang berbeda. Maka kita bisa memanggil method ini pada class main dengan cara sebagai berikut:

```

public class Main {
    public static void main(String[] args) {
        // membuat object dengan 2 argumen
        Superman superman = new Superman("Ken", 22);

        // memanggil method yang dilakukan overloading
        superman.attack("Creep", 13);
        superman.attack("Hulk");
        superman.attack();
    }
}

```

Ketika kita jalankan programnya maka akan output seperti ini:

```
Superman menyerang Creep sebanyak 13
Superman menyerang Hulk
Superman menyerang musuh
DC: DUNIA SUPERHEROES
```

Manfaat menggunakan **Overloading Method** adalah:

- **Mempermudah penggunaan metode** dengan berbagai parameter yang berbeda
- **Meningkatkan keterbacaan kode** dengan menggunakan satu nama metode untuk berbagai operasi yang mirip
- **Mendukung fleksibilitas** dalam pemrograman tanpa harus membuat metode dengan nama berbeda untuk setiap variasi parameter

### **Dynamic Polymorphism (overriding method)**

**Dynamic polymorphism (polimorfisme dinamis)** dapat kita lakukan dengan cara menerapkan **method overriding** yang telah kita pelajari di modul sebelumnya. Hal ini bisa dilakukan ketika kita **membuat sebuah object dengan tipe parent class tetapi memanggil constructor dari child class atau juga pemanggilan method-nya**. Contoh kita memiliki class Hero seperti ini:

```
public class Hero {
    public void melindungi(){
        System.out.println("Hero melindungi masyarakat");
    }
}
```

Dan terdapat class Superman seperti ini:

```
public class Superman extends Hero{
    @Override
    public void melindungi(){
        System.out.println("Superman melindungi masyarakat bumi dari serangan monster");
    }
}
```

Coba kita buat object dari class Superman dan memanggil method-nya dengan cara berikut:

```
public class Main {
    public static void main(String[] args) {
        Hero superman = new Superman(); // Type Hero tapi constructor dari child class Superman
        superman.melindungi(); // yang pertama

        superman = new Hero(); // Type Hero dan constructor dari Hero juga (parent class)
        superman.melindungi(); // yang kedua
    }
}
```

Jika kita coba jalankan programnya maka akan output seperti berikut:

```
Superman melindungi masyarakat bumi dari serangan monster
Hero melindungi masyarakat
```

Penjelasan:

- Dari kode Main class, kita membuat sebuah objek **superman** dengan tipe data **Hero**, tetapi **referensi class** adalah Superman.
- Pada saat pemanggilan method **melindungi()** dipanggil yang pertama, method yang dieksekusi adalah method **melindungi()** yang ada di dalam class **Superman**.
- Setelah itu objek superman dibuat referensinya ke Hero yaitu kode **superman = new Hero()**.
- Pada pemanggilan method melindungi yang kedua, method yang dieksekusi adalah method **melindungi()** yang berada di class **Hero**.

Hal ini juga disebut sebagai **Virtual Method Invocation**, disebut virtual karena antara method yang dikenali oleh **compiler** dan method yang dijalankan oleh **JVM** berbeda. Saat compile time, compiler akan mengenali method **melindungi()** yang akan dipanggil adalah method **melindungi()** yang ada di class **Hero**, karena objek bertipe **Hero**. Tetapi saat dijalankan (**runtime**), maka yang dijalankan oleh **JVM** adalah method **melindungi()** yang ada di class **Superman**. Tipe data sebelah kiri atau sebelum variabel juga dapat berupa interface (materi ada di akhir modul). Secara kode akan tampak seperti ini:

```
Interface namaObjek = new classYangDiimplement();
```

## Object Casting

Sebelum masuk ke **object casting** kita harus pelajari terlebih dahulu tentang **type casting** di java, yaitu **mengubah tipe data** ke tipe data yang lain. Di java juga terdapat dua macam tipe casting yaitu:

- **Widening casting** (otomatis) – mengkonversi tipe data yang lebih kecil ke yang lebih besar

`byte -> short -> char -> int -> long -> float -> double`

Contoh:

```
public class Main {
    public static void main(String[] args) {
        int myInt = 9;
        double myDouble = myInt; // Otomatis dikonversi ke double dari int

        System.out.println(myInt);      // Outputs 9
        System.out.println(myDouble);   // Outputs 9.0
    }
}
```

- **Narrowing casting** (manual) – mengkonversi tipe data yang lebih besar ke yang lebih kecil

`double -> float -> long -> int -> char -> short -> byte`

Contoh:

```
public class Main {
    public static void main(String[] args) {
        double myDouble = 9.78d;
        int myInt = (int) myDouble;
        /* Konversi manual dari double ke int dengan cara
           menggunakan tanda kurung "(tipeData) nilai double" */

        System.out.println(myDouble);    // Outputs 9.78
        System.out.println(myInt);      // Outputs 9
    }
}
```

**Type casting** berbeda dengan **object casting**, ketika kita menggunakan **type casting** kita cukup menggunakan (**data type**) *nama variabel*. Sedangkan untuk **object casting** yang diubah adalah **Reference Data Type**. Superman, Spiderman, dan Deadpool yang sudah kita buat sebelumnya bisa kita gunakan sebagai **tipe data referensi**. **Object casting** disini berperan untuk mengubah tipe data dari **subclass** ke **superclass (upcasting)** atau sebaliknya (**downcasting**). Untuk contoh kita akan buat seperti ini:

File: Hero.java

```
public class Hero {  
    public void melindungi(){  
        System.out.println("Hero melindungi masyarakat");  
    }  
}
```

File: Spiderman.java

```
public class Spiderman extends Hero{  
    @Override  
    public void melindungi(){  
        System.out.println("Spiderman melindungi masyarakat");  
    }  
}
```

File: Superman.java

```
public class Superman extends Hero{  
    @Override  
    public void melindungi(){  
        System.out.println("Superman melindungi masyarakat bumi dari serangan monster");  
    }  
}
```

File: Deadpool.java

```
public class Deadpool extends Hero{  
    @Override  
    public void melindungi(){  
        System.out.println("Deadpool melindungi masyarakat");  
    }  
}
```

Untuk contoh **upcasting**:

```
public class Main {  
    public static void main(String[] args) {  
        Superman superman = new Superman();  
  
        // kode berikut upcasting dari child ke parent class  
        Hero hero = superman;  
    }  
}
```

Untuk contoh **downcasting**:

```
public class Main {  
    public static void main(String[] args) {  
        Hero hero = new Hero();  
  
        // kode berikut downcasting  
        Superman superman = (Superman) hero;  
    }  
}
```

Untuk proses **downcasting** kita harus menulis explisit pada class apa kita ingin mengubah parent class ke child class dengan ditulis tanda kurung (**childclass**) sebelum objek yang akan di-casting. Sebutan lain untuk **downcasting** adalah **explicit casting**, sedangkan untuk **upcasting** sebutan lainnya adalah **implicit casting**.

### Heterogeneous Collection

Pada **pemrograman dasar** kita sudah mempelajari apa itu sebuah **array**. Dengan adanya konsep **polimorfisme**, maka **variabel array** bisa dibuat **heterogen**. Yang artinya di dalam array tersebut bisa berisi **berbagai macam objek** yang berbeda tetapi dengan memperhatikan bahwa objek tersebut masih memiliki relasi yang sama terhadap parent class. Kita bisa menyimpan objek dari class Superman, Spiderman, Deadpool pada sebuah array dengan tipe data parent class. Contohnya adalah seperti ini:

```
public class Main {
    public static void main(String[] args) {
        // pembuatan hero dengan type class Hero
        Hero[] heros = new Hero[3];

        // pengisian nilai array Hero dengan child class
        heros[0] = new Spiderman();
        heros[1] = new Superman();
        heros[2] = new Deadpool();

        // coba panggil di Looping
        for (Hero hero : heros) {
            hero.melindungi();
        }
    }
}
```

Output program:

```
Spiderman melindungi masyarakat
Superman melindungi masyarakat bumi dari serangan monster
Deadpool melindungi masyarakat
PS E:\KULIAH\ASILAB\MODUL_PBO\polyAdhoc>
```

Pada kode di atas, kita mengumpulkan beberapa child class ke dalam satu variabel yang sama dengan tipe data parent class. Data ke-1 berisi objek **Spiderman**, data ke-2 berisi objek **Superman**, data ke-3 berisi objek **Deadpool** yang ketiganya merupakan turunan dari class **Hero**. Kemudian kita memanggil method **melindungi()** dengan menggunakan **looping foreach**, hal ini tidak menimbulkan error karena semua objek yang tersimpan di array **Hero** memiliki method **melindungi()**.

Untuk menggunakan looping **for** biasa:

```
for (int i = 0; i < heros.length; i++){
    heros[i].melindungi();
}
```

## Abstraction

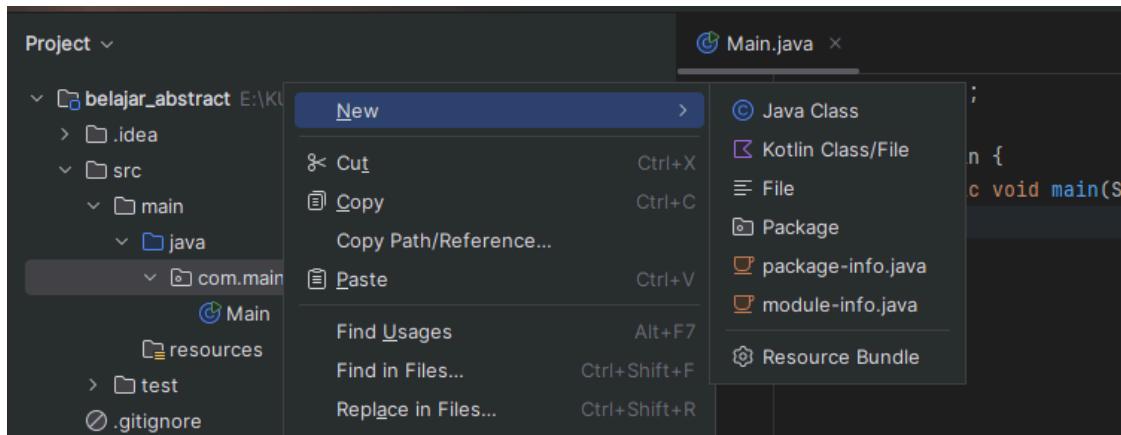
**Abstract** adalah suatu ide yang bukan objek materi (**tidak memiliki bentuk fisik**), lawan kata **abstract** adalah **concrete**. Dalam konteks PBO, **abstraction** adalah konsep yang memungkinkan kita untuk **menyembunyikan kompleksitas** dari sebuah objek dan hanya menampilkan fungsionalitas yang penting dalam sebuah interaksi.

### Abstract Class

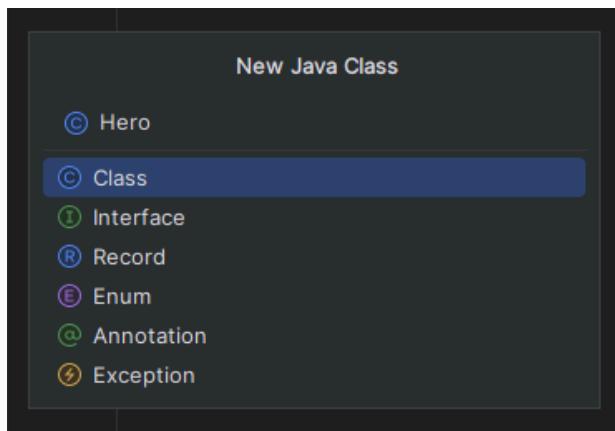
**Abstract class** adalah kelas yang dideklarasikan dengan keyword **abstract**, dan sifatnya tidak dapat dibuat menjadi suatu objek (**tidak dapat diinstansiasi**) karena bentuknya memang belum jelas (*abstract*). **Abstract class** biasanya digunakan untuk **generalisasi objek** yang sangat umum, misalnya Hewan, Kendaraan, *Database*, dan lain sebagainya.

**Abstract class** berguna untuk **diturunkan / diwarisi** kelas lainnya menggunakan keyword **extends**, penggunaan utamanya adalah untuk **polimorfisme (Polymorphism)** yang sudah dipelajari di atas. Dalam suatu abstract class dapat berisi atribut (variabel), **abstract method** maupun **concrete method**.

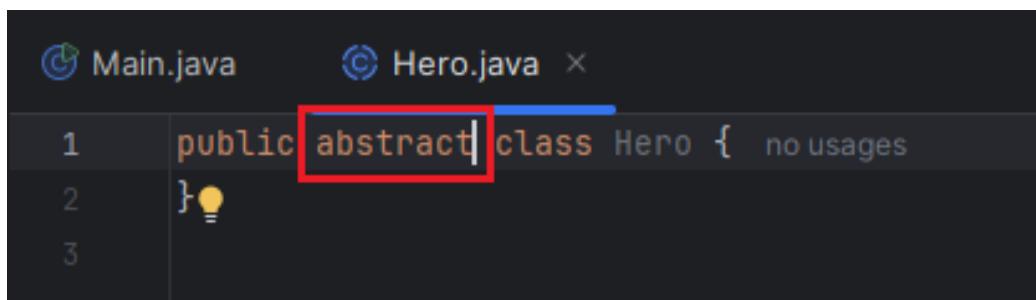
Analogi mengenai class abstract ialah seperti ini, ketika mendengar sebuah kata “**hero**” tentu yang dipikirkan pertama kali adalah “hero apa yang dimaksud? assasin, marksman, mage, tank atau apa?” apapun itu, semuanya adalah **hero**. Kita tahu bahwa semua **hero** pasti bisa **attack**, tapi bagaimana kalau “hero menyerang” hal ini akan menimbulkan sebuah pertanyaan hero apa yang dimaksud. Inilah yang disebut sebagai **abstraksi**, kata “**hero**” sendiri masih bersifat **abstract**. Untuk cara membuat sebuah class abstract, seperti membuat sebuah class biasa:



Isi nama dan pilih **Class**



Setelah class hero dibuat oleh intelliJ IDEA, bisa tambahkan kata kunci *abstract* setelah *modifier* class, seperti berikut:



Ketika kita sudah membuat sebuah class ***abstract***, maka class tersebut tidak bisa dibuat sebuah **instance objek**. Error berikut akan muncul ketika kita mencoba untuk membuat sebuah **instance objek** dari sebuah class yang bersifat **abstrak**.

```

1 package com.main;
2
3 public class Main {
4     public static void main(String[] args) {
5         Hero hero = new Hero();
6     }
7 }

```

'Hero' is abstract; cannot be instantiated

Make 'Hero' not abstract Alt+Shift+Enter More actions... Alt+Enter

com.main  
belajar\_abstract

Sebuah class yang berbentuk **abstrak** hanya bisa kita pakai ketika memiliki sebuah **inheritance** dari class tersebut, contoh kita buat sebuah class **Assasin.java** yang akan menjadi **child class** dari **Hero**.

```

1 package com.main;
2
3 public class Assasin extends Hero{ no usages }
4
5 }
6

```

Maka sekarang kita bisa membuat sebuah **objek** dengan tipe **Hero**, tetapi menggunakan reference dari **Assasin**.

```

1 package com.main;
2
3 public class Main {
4     public static void main(String[] args) {
5         Hero hero = new Assasin();
6     }
7 }

```

Sebuah **class abstract** dibuat untuk mengumpulkan sebuah method umum atau sebuah ide suatu class berperilaku. Sebagai contoh jika kita membuat sebuah class hero Assasin, Mage, Tank dan secara umum hero-hero tersebut bisa menyerang,

kembali ke *base*, menggunakan item dengan cara menyerang dan menggunakan item yang berbeda-beda, maka dari itu bisa dikumpulkan dalam satu *class* yang bernama **Hero** tetapi class tersebut **tidak bisa dipakai secara langsung** melainkan dipakai melalui **child class**.

### Abstract Method

**Abstract Method** adalah method yang bersifat **abstrak** yang ditandai dengan penambahan keyword **abstract** pada deklarasinya serta **tidak memiliki implementasi body / isi**. Deklarasi *abstract method* langsung diakhiri tanda titik koma (;), tanpa tanda kurung kurawal ({...}). Semua *abstract method* yang ada akan dipaksakan untuk ada di kelas turunannya (**harus di-override**).

**Abstract method** dipakai ketika sebuah class memiliki **method yang sama tetapi perlakunya berbeda**. Seperti hero assassin dan hero tank pasti bisa menyerang, tetapi cara menyerang hero assassin dan hero tank berbeda. Maka dari itu diperlukan sebuah method di parent class tetapi method itu diisi kosong yang akan disesuaikan dengan masing-masing hero nantinya.

Berikut adalah cara membuat sebuah *abstract method*:

```
public abstract class Hero {
    public abstract void namaMethod();
}
```

Untuk bentuk perbedaan antara method biasa dengan method abstrak:

```
public abstract class Hero {
    // method abstract
    public abstract void namaMethod();

    // method biasa atau concrete
    public void namaMethod2(){
        // method ini memiliki body
    }
}
```

Hal yang perlu diperhatikan ketika membuat sebuah method abstract, class dimana method itu dibuat **harus berbentuk abstrak juga**. Seperti contoh kode di atas, bentuk class **Hero** harus abstrak jika tidak maka akan terjadi eror.

### **Contoh Implementasi Abstraction**

Pada dasarnya setiap **objek** yang berbeda sering memiliki **kesamaan** atau **kemiripan** tertentu. Kita ambil contoh kelas hero di game mobile legend. Misalnya, tank dan marksman memiliki kemiripan yaitu sama-sama bisa melakukan **attack**. Tetapi **hero** adalah hal yang sangat umum. **Hero** sendirinya bukanlah suatu objek. Tentu saat kita menyebutkan '**hero attack**' kita tidak tahu **hero** tersebut apa karena ia tidak merujuk kepada hal yang lebih **spesifik**. Oleh karena itu, hero dapat diubah menjadi **kelas abstrak**.

Pertama-tama dalam projek anda buatlah kelas **Main** pada file **Main.java** yang berisi **main** method:

```
public class Main {
    public static void main(String[] args) {

    }
}
```

Kemudian buatlah kelas **Hero** pada file **Hero.java**

```
public abstract class Hero {
    // method abstract
    public abstract void attack();
}
```

Untuk membuktikan bahwa kelas **Hero** tidak dapat dijadikan objek akan kita coba instansiasi kelas **Hero** di kelas **Main**.

```
src > Main.java > Main
1 public class Main {
2     Run | Debug
3     public static void main(String[] args) {
4         Hero hero = new Hero();
5     }

```

Cannot instantiate the type Hero Java(16777373)  
Fix in Chat (Ctrl+Shift+D)  
Ctrl+click to open in new tab  
Hero  
View Problem (Alt+F8) Quick Fix... (Ctrl+.)

Terbukti, akan muncul error “**Cannot instantiate the type Hero**” dikarenakan kelas **Hero** adalah **abstract class**.

Kita akan membuat kelas lain, **Tank** pada file **Tank.java** dan **Marksman** pada file **Marksman.java** yang masing-masing akan **meng-extends** kelas **Hero**.

Perhatikan apa yang terjadi saat awal kita **meng-extends** class **Hero**, akan muncul error merah pada kode yang dibuat:

```
src > Tank.java > Tank
1 public class Tank extends Hero {
```

Dengan error “**The type Tank must implement the inherited abstract method Hero.attack()**”.

Di VSCode :

```
src > Tank.java > Tank
1 public class Tank extends Hero {
```

The type Tank must implement the inherited abstract method Hero.attack() Java(67109264)  
Fix in Chat (Ctrl+Shift+D)  
Ctrl+click to open in new tab  
Tank  
View Problem (Alt+F8) Quick Fix... (Ctrl+.)

Di IntelliJ IDEA :

```
1 public class Tank extends Hero { no usages
2
3 }

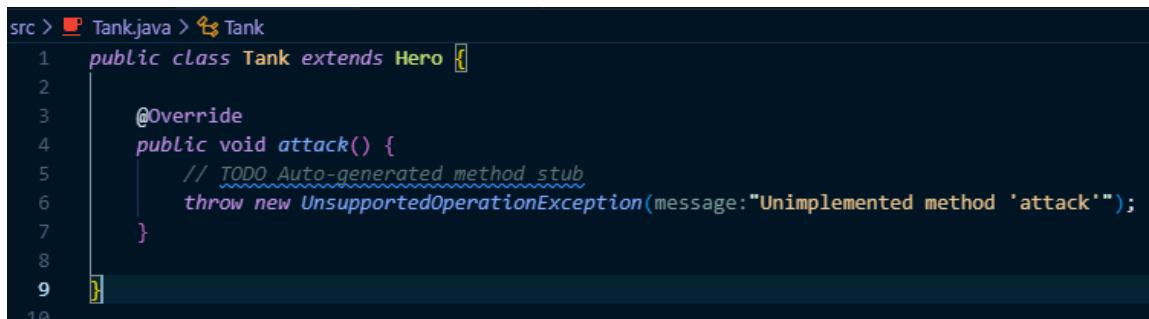
```

Class 'Tank' must either be declared abstract or implement abstract method 'attack()' in 'Hero' :  
Implement methods Alt+Shift+Enter More actions... Alt+Enter

```
public class Tank
extends Hero
polyAdhoc
```

Apa yang terjadi? kelas Tank harus mengimplementasikan abstract method yang ada pada abstract class Hero yaitu method attack. Mari kita coba perbaiki, caranya bisa diketik atau secara otomatis di VSCode kalian bisa meng-klik Quick Fix (Ctrl+) seperti di atas atau di IDE lain cari 💡 kemudian klik **add unimplemented method**. IDE akan otomatis megenerate.

Perhatikan anotasi **@Override** sebelum **method** attack, hal ini diperlukan karena class **Tank** merupakan turunan dari class **Hero** yang juga memiliki method **attack** meskipun method tersebut bersifat **abstract**.



```
src > Tank.java > Tank
1 public class Tank extends Hero {
2
3     @Override
4     public void attack() {
5         // TODO Auto-generated method stub
6         throw new UnsupportedOperationException(message:"Unimplemented method 'attack'");
7     }
8
9 }
10
```

Akan kita isi method attack di atas dengan skill yang dimiliki mage.

```
public class Tank extends Hero{

    @Override
    public void attack() {
        System.out.println("Mage menyerang dengan menggunakan armor shield!");
    }
}
```

Lakukan yang sama pada kelas Marksman dengan skillnya.

```
public class Marksman extends Hero{

    @Override
    public void attack() {
        System.out.println("Marksman menyerang dengan 5000 damage!");
    }
}
```

Kemudian kita bisa membuat objek dari kelas Tank dan Marksman.

```

public class Main {
    public static void main(String[] args) {

        Tank tank = new Tank();
        Marksman marksman = new Marksman();

        tank.attack();
        marksman.attack();
    }
}

```

Coba jalankan kode tersebut apa yang terjadi? Kalau benar, akan muncul output seperti berikut sesuai urutan pemanggilan method pada objek.

```

Mage menyerang dengan menggunakan armor shield!
Marksman menyerang dengan 5000 damage!
[REDACTED]

```

Apa yang dapat kita simpulkan? **Tank** dan **Marksman** sama-sama **Hero** dan keduanya dapat melakukan **attack**, tetapi tentu saja **attack** dari keduanya **berbeda**.

#### TIPS

[Video](#)

[Materi](#)

## Interface

Masih berkaitan dengan **abstraction**. **Interface** adalah satu cara untuk mencapai **abstraction** secara **menyeluruh (total abstraction)**. Namun, **interface** bukan **class**, meskipun terlihat sama. **Interface** digunakan untuk mendefinisikan suatu **sifat-sifat (behaviours, berupa method)** suatu class.

Persamaannya dengan **abstract class** adalah keduanya **tidak dapat diinstansiasi** menjadi **objek**, lantas apa perbedaannya?

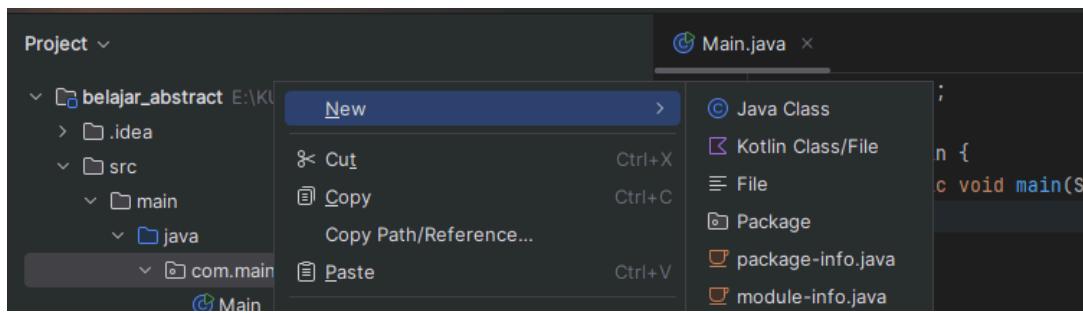
Abstract Class	Interface
Bisa memiliki abstract & concrete method.	Hanya abstract method.
Bisa memiliki method static dan final.	Method tidak boleh bersifat static dan final.
Access modifier perlu ditulis sendiri.	<b>Secara implisit</b> semua method adalah public abstract.
Bisa memiliki <i>constants</i> dan <i>instance variables</i> .	Hanya dapat memiliki <i>constants</i> karena secara implisit semua variabel dalam interface adalah public static final.
Hanya dapat meng- <b>extends</b> <u>satu</u> abstract class lainnya (tidak bisa multiple inheritance).	Dapat meng- <b>implements</b> lebih dari 1 interface.
Dapat meng- <b>implements</b> interface lebih dari satu interface.	Tidak dapat meng- <b>implements</b> interface lain.
Bisa membuat properti atau variabel	Hanya bisa buat konstanta saja

Secara sederhana **interface** digunakan untuk **memaksa** sebuah **class** untuk memakai sebuah **method** yang sebelumnya **sudah ditentukan**. Misalnya, **burung** adalah **hewan** dan **burung** bisa **terbang**, **pesawat** juga bisa **terbang**, lantas apakah **pesawat** adalah **hewan**? tentu **bukan**, keduanya memiliki **sifat / behaviour** sama-sama bisa **terbang**, kita dapat membuat interface **Flyable** untuk kedua **objek** tersebut. Penggunaan **interface** tidak selalu seperti di atas, selebihnya tentunya tergantung **use-case**.

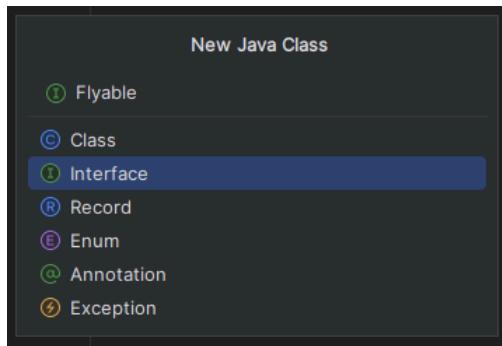
Contoh pendeklarasian *interface*:

```
// bisa diisi dengan variabel
interface Flyable {
    // secara default sudah public
    void terbang();
}
```

Cara pembuatan sebuah interface sama dengan cara membuat class biasa, seperti berikut:



Pilih *Interface* dan isi nama interface yang akan dibuat



Dalam kasus ini kita akan membuat sebuah kemampuan untuk **terbang** pada class **Hewan** dan juga class **Pesawat**. Karena kedua **class** tersebut **tidak bisa berasal dari parentclass yang sama**, maka kita tidak bisa untuk membuat **method abstract**, yang bisa kita lakukan adalah membuat sebuah *Interface* yang nantinya bisa digunakan untuk memaksa class **Burung** dan class **Pesawat** harus **mengimplementasikan** sebuah method untuk **terbang**.

Setelah kita membuat sebuah interface **Flyable** sebelumnya, kita bisa membuat sebuah method yang dibutuhkan ke dalam **interface**. Dalam kasus ini kita membutuhkan method **terbang**, maka bisa kita isi *Interface Flyable* menjadi seperti berikut:

```
public interface Flyable {
    void terbang();
}
```

Untuk cara memakai sebuah *Interface* yang sudah kita buat, bisa gunakan sebuah kata kunci **implements** pada class yang dituju. Contohnya **interface Flyable** akan diimplementasikan ke class **Pesawat** dan **Burung**, maka kode dari kedua class akan menjadi seperti ini:

## File Burung.java

```
package com.main;

public class Burung implements Flyable{ no usages }
```

Class 'Burung' must either be declared abstract or implement abstract method 'terbang()' in 'Flyable' :  
Implement methods Alt+Shift+Enter More actions... Alt+Enter

com.main

## File Pesawat.java

```
package com.main;
public class Pesawat implements Flyable{ no usages }
```

Class 'Pesawat' must either be declared abstract or implement abstract method 'terbang()' in 'Flyable'  
Implement methods Alt+Shift+Enter More actions... Alt+Enter

com.main

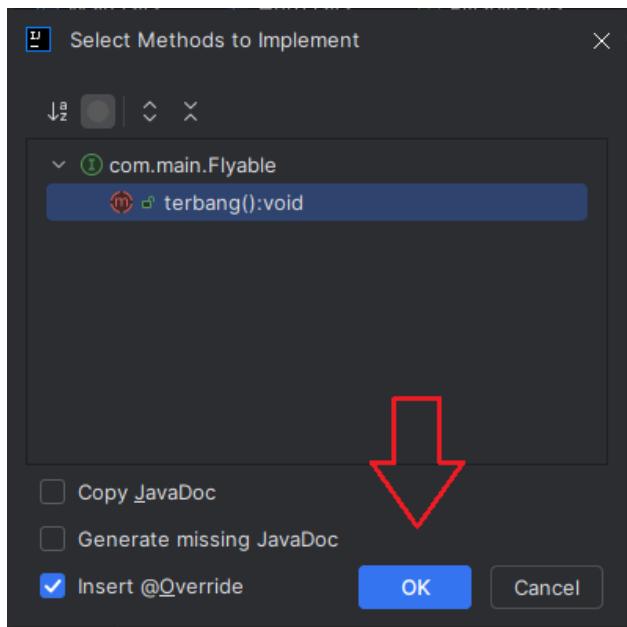
Ketika melakukan **implements** dari suatu **Interface** ke dalam sebuah class, akan langsung muncul garis merah yang menandakan terjadi sebuah error dengan pesan “**The type Pesawat must implement the inherited abstract method Flyable.terbang()**” yang artinya kita melakukan sebuah **implementasi** dari sebuah **interface** tetapi kita tidak melakukan **override** terhadap **method** yang ada di dalam **Interface** yang bersangkutan. Maka solusinya kita bisa arahkan **cursor** ke kode yang **error**, lalu klik **Implement methods**.

```
public class Pesawat implements Flyable{ no usages }
```

Class 'Pesawat' must either be declared abstract or implement abstract method 'terbang()' in 'Flyable'  
Implement methods Alt+Shift+Enter More actions... Alt+Enter

com.main

Setelah itu bisa di klik OK:



Maka kode akan berubah tidak error seperti ini:

```
public class Pesawat implements Flyable{

    @Override
    public void terbang() {

    }
}
```

Setelah itu bisa kita sesuaikan isi dari method **terbang()** pada masing-masing class, dalam hal ini di dalam class **Burung** method **terbang** menjadi seperti ini:

```
@Override
public void terbang() {
    System.out.println("Burung terbang menggunakan sayap");
}
```

Pada class Pesawat method terbang seperti ini:

```
@Override
public void terbang() {
    System.out.println("Pesawat terbang menggunakan mesin");
}
```

Bisa kita buat sebuah objek di dalam Main class dengan cara seperti berikut:

```
public class Main {
    public static void main(String[] args) {
        // membuat objek seperti biasa
        Burung burung = new Burung();
        Pesawat pesawat = new Pesawat();

        // membuat objek dengan polymorphism
        Flyable burung1 = new Burung();
        Flyable pesawat1 = new Pesawat();

        //memanggil method pada objek burung
        burung.terbang();
        burung1.terbang();

        // memanggil method pada objek pesawat
        pesawat.terbang();
        pesawat1.terbang();
    }
}
```

Ketika dijalankan maka akan keluar sebuah output seperti ini:

```
Burung terbang menggunakan sayap
Burung terbang menggunakan sayap
Pesawat terbang menggunakan mesin
Pesawat terbang menggunakan mesin
PS-E:\KULIAH\SEMESTER 3\POLY\polyAdbeck
```

Setelah pembahasan ini mungkin akan ada sebuah pertanyaan yang muncul yaitu “Apa perbedaan spesifik antara interface dan sebuah class abstract dalam java?”. Jawabannya ialah **Interface** digunakan ketika kita ingin mengimplementasikan sebuah method tertentu

yang digunakan dengan **inheritance** tidak memungkinkan karena berbeda parent class dan juga **Interface** digunakan dalam sebuah kasus kita ingin melakukan extends lebih dari 1 parent class yang dimana sebuah **extends** dalam java hanya bisa satu parent, maka dari itu dibuatlah sebuah **Interface**.

### **Is-a relation & Operator instanceof**

Di Java kita bisa melihat suatu objek adalah suatu **instance** dari **class** apa, dengan operator **instanceof** yang akan mengembalikan nilai **boolean**. Menggunakan potongan kode sebelumnya yaitu Hero, Marksman, dan Tank. Silahkan coba code berikut:

```
public class Main {
    public static void main(String[] args) {
        Tank tank = new Tank();

        System.out.println("Apakah tank is-a Object = " + (tank instanceof Object));
        System.out.println("Apakah tank is-a Hero = " + (tank instanceof Hero));
        System.out.println("Apakah tank is-a Tank = " + (tank instanceof Tank));
    }
}
```

Output dari kode di atas adalah berikut:

```
Apakah tank is-a Object = true
Apakah tank is-a Hero = true
Apakah tank is-a Tank = true
```

Namun, coba tambahkan pada baris baru kode berikut:

```
System.out.println("Apakah tank is-a Marksman = " + (tank instanceof Marksman));
```

Bahkan sebelum kalian **run** IDE kalian akan menunjukkan kepada kalian bahwa kode tersebut akan **error**. Jika tetap kalian **run**, **compiler** akan menampilkan error “**Incompatible conditional operand types Tank and Marksman**”. Ini dikarenakan **Tank** meskipun sebuah **Hero** tetapi ia bukanlah **Marksman**.

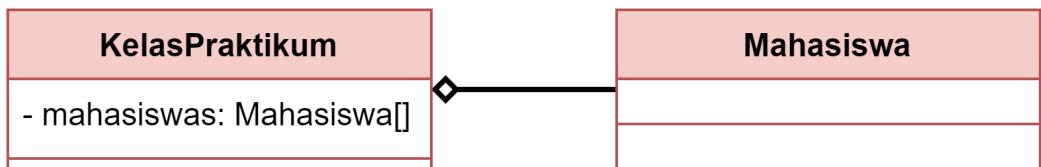
Maksud dari **is-a** adalah untuk mengecek apakah sebuah object merupakan **inheritance** dari sebuah **class** tertentu. Pada kode di atas kita mengecek apakah **tank** **inheritance** dari class **Object**, **Hero**, dan **Tank**. Ketika kode tersebut mengembalikan

sebuah nilai **true**, itu artinya objek yang dibuat memiliki hubungan dengan class yang bersangkutan.

### Has-a relation

Dalam PBO, konsep "**has-a**" relation atau **agregasi (aggregation)** mengacu pada hubungan antara dua kelas, di mana suatu objek dari kelas A memiliki satu atau lebih objek dari kelas B sebagai member atau atribut dari dirinya.

Misalnya, Kelas Praktikum memiliki (banyak) Mahasiswa. Jika digambarkan dalam class diagram adalah sebagai berikut:



Atau dalam kode Java:

Mahasiswa.java

```

public class Mahasiswa {
    String nim, nama;

    public Mahasiswa(String nim, String nama){
        this.nim = nim;
        this.nama = nama;
    }
}
  
```

KelasPraktikum.java

```

public class KelasPraktikum {
    String name;
    Mahasiswa mahasiswa;

    public KelasPraktikum(String name, Mahasiswa mahasiswa){
        this.name = name;
        this.mahasiswa = mahasiswa;
    }
}
  
```

### Main.java

```
public class Main {
    public static void main(String[] args) {
        Mahasiswa mahasiswa = new Mahasiswa("202310370311006", "Ken Aryo Bimantoro");
        KelasPraktikum kelasPraktikum = new KelasPraktikum("PBO A", mahasiswa);

        System.out.println("Kelas praktikum " + kelasPraktikum.name + " memiliki mahasiswa "
            + kelasPraktikum.mahasiswa.nama);
    }
}
```

Output program:

Perhatikan objek **mahasiswa** menjadi bagian dari objek **kelasPraktikum**. Dengan demikian **KelasPraktikum** *has-a* (memiliki sebuah) **Mahasiswa**.

### Relasi lain

Relationship	UML Connector
Inheritance	—→
Interface inheritance	- - - - - →
Dependency	- - - - - →
Aggregation	◇ —————
Association	————
Directed association	————→

Sebenarnya masih ada relasi lain dalam class diagram, dengan pengantar 2 relasi di atas, mungkin tanpa disadari saat Anda mengerjakan projek Java, Anda sudah mengimplementasikan relasi yang lain. Untuk memahami teori relasi yang lain, silakan Anda berselancar di internet.

### TIPS

Materi tambahan tentang Interface di Java:

[Video](#)

[Materi](#)

Materi tambahan tentang Polymorphism di Java:

[Video](#)

[Materi](#)

### PRAKTEK

Sekarang mari coba terapkan konsep Package dan Interface dalam projek praktek yang sudah kita kerjakan dari modul modul sebelumnya. Sebelum mulai, mari kita jawab pertanyaan ini:

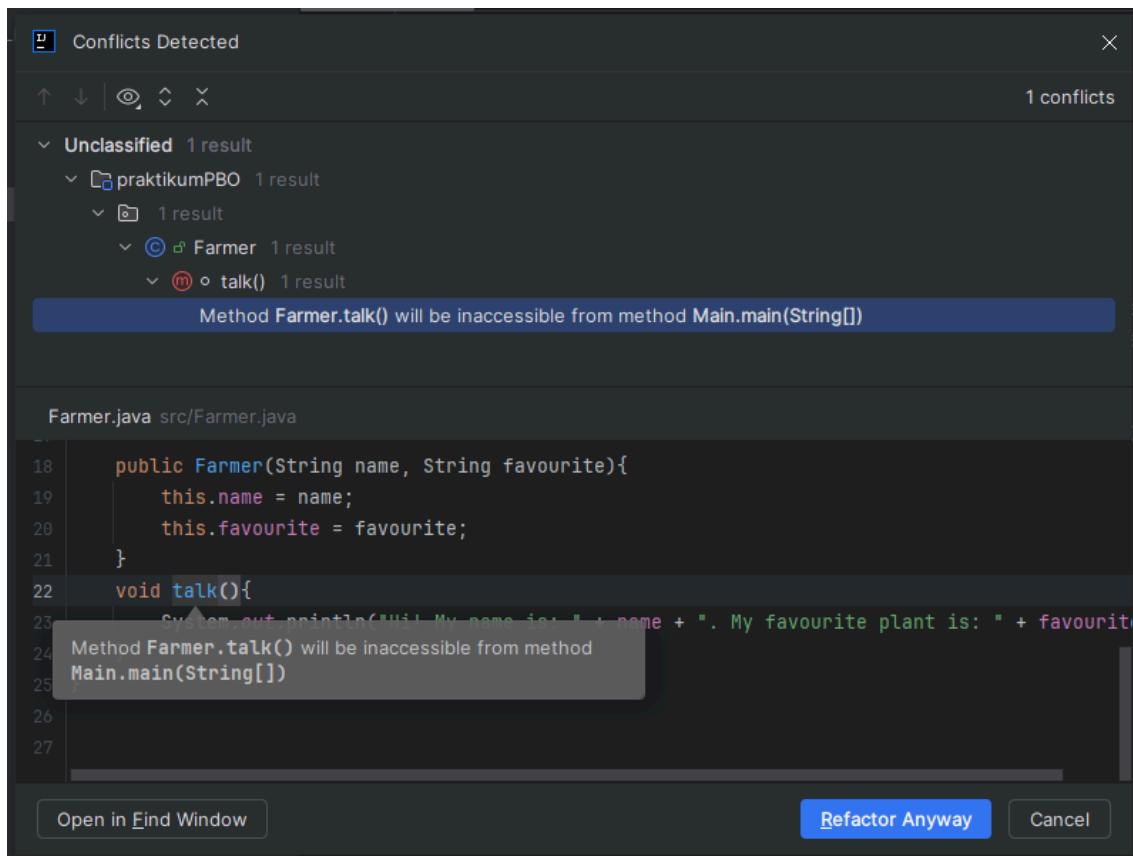
"Bagaimana kita bisa mengorganisir kode kita lebih baik dengan package dan menambahkan fleksibilitas dengan interface?"

Benar! Kita bisa:

- Mengelompokkan class-class terkait ke dalam package
- Membuat interface untuk mendefinisikan perilaku umum yang bisa diimplementasikan oleh berbagai class

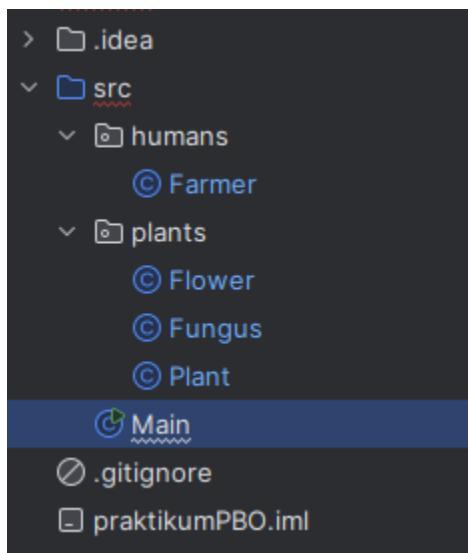
#### Langkah 1: Membuat Package

1. Klik kanan pada folder **src** → **New → Package**
2. Beri nama package “**plants**” (untuk class **Plant**, **Flower**, **Fungus**) dan “**humans**” (untuk **Farmer**)



Jika menemui seperti ini, bisa klik “**Refactor Anyway**” saja.

Maka struktur project akan menjadi:



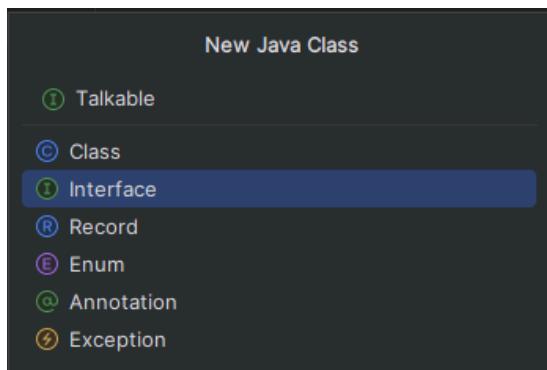
Jangan lupa untuk menambahkan deklarasi package di awal setiap file:

```
// Di Farmer.java  
package humans;  
  
// Di Plant.java, Flower.java, Fungus.java  
package plants;
```

## Langkah 2: Membuat Interface

Buat interface **Talkable** untuk mengatur perilaku class yang bisa '**berbicara**':

1. Klik kanan pada folder **plants** → **New** → **Java Class -> Interface**
2. Lalu ketikkan **Talkable**



Dan isikan seperti berikut pada file tersebut:

```
package plants;  
  
public interface Talkable {  
    void talk(); // Method interface (tanpa implementasi)  
}
```

Modifikasi class **Plant** untuk mengimplementasikan interface tersebut:

```

package plants;

public class Plant implements Talkable {
    protected String name;

    // ... constructor dan method lainnya ...

    @Override
    public void talk() {
        System.out.println("I am a plant named " + name);
    }
}

```

### Langkah 3: Menerapkan Polymorphism dan Overloading

Tambahkan method **overload** di class **Farmer**:

```

package humans;

import plants.Talkable;

public class Farmer {
    private String name;
    private String favourite;

    // Constructor sebelumnya
    public Farmer(String name, String favorite) { ... }

    // Overload constructor
    public Farmer(String name) {
        this.name = name;
        this.favourite = "No favorite plant yet";
    }

    // Method overload untuk talk()
    public void talk(Talkable talkable) {
        talkable.talk();
    }

    // ... method lainnya ...
}

```

#### Langkah 4: Menggunakan Abstract Class

Buat abstract class **LivingThing** di package **plants**:

```
package plants;

public abstract class LivingThing {
    protected String name;

    public abstract void grow(); // Abstract method

    public void breathe() {
        System.out.println(name + " is breathing");
    }
}
```

Modifikasi class **Plant** untuk mewarisi **abstract** class:

```
package plants;

public class Plant extends LivingThing implements Talkable{
    public String getName(){
        return name;
    }
    public void setName(String name){
        this.name = name;
    }
    public Plant(String name){
        this.name = name;
    }
    @Override
    public void talk() {
        System.out.println("I am a plant named " + name);
    }
    @Override
    public void grow() {
        System.out.println(name + " is growing slowly");
    }
}
```

### Langkah 5: Implementasi di Main Class

```

import plants.*;
import humans.Farmer;

public class Main {
    public static void main(String[] args) {
        // Pakai constructor overload
        Farmer farmer3 = new Farmer("New Farmer");

        // Polymorphism dengan interface
        Talkable myPlant = new Flower("Rose");
        farmer3.talk(myPlant); // Akan memanggil talk() dari Flower

        // Abstract method
        Plant cactus = new Plant("Cactus");
        cactus.grow(); // Output: "Cactus is growing slowly"
        cactus.breathe(); // Output: "Cactus is breathing"
    }
}

```

### Output yang diharapkan:

```

C:\Users\User\.gradle\jdk8\bin>java Main
Hi! I'm Rose! And I'm a Flower!
Cactus is growing slowly
Cactus is breathing
Process finished with exit code 0

```

### Kesimpulan Praktek:

1. Package membantu mengorganisir kode
2. Interface (**Talkable**) memungkinkan polymorphism
3. Overloading constructor memberi fleksibilitas inisialisasi object
4. Abstract class (**LivingThing**) memaksa child class untuk mengimplementasikan method tertentu

### Coba eksplorasi dengan menambahkan:

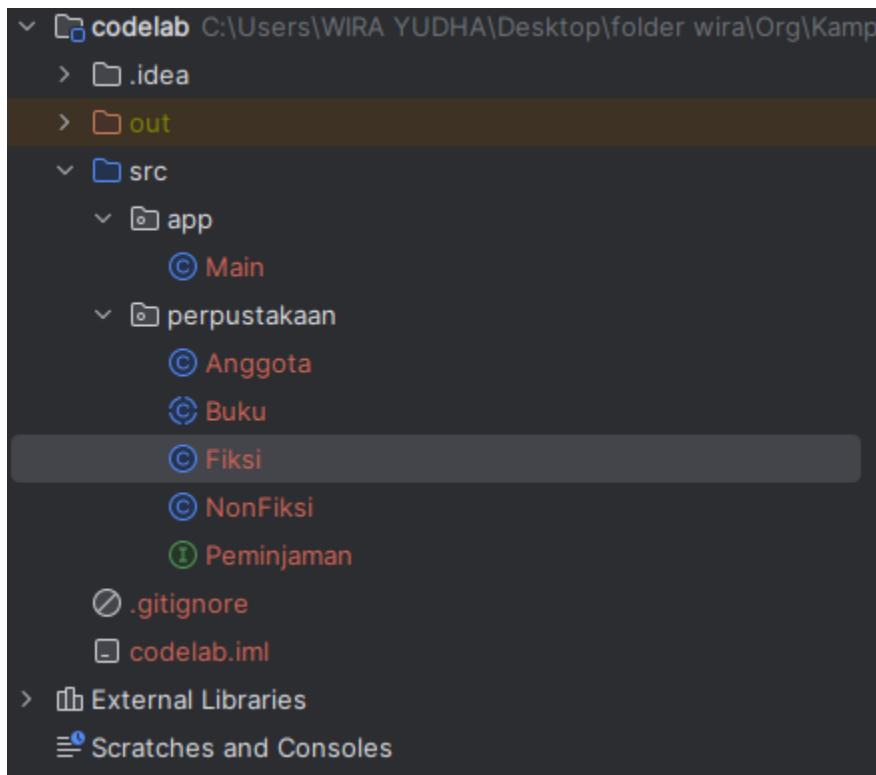
- Interface baru seperti Growable
- Abstract method lainnya
- Package baru untuk class-class tambahan

### **CODELAB & TUGAS**

#### **CODELAB**

Buatlah sistem manajemen perpustakaan sederhana yang menerapkan konsep-konsep dasar pemrograman berorientasi objek dalam Java, yaitu **Package, Polymorphism, Overloading, Interface, dan Abstraction.**

1. Semua kelas harus disimpan dalam package perpustakaan (kecuali **Main.java**). Berikut adalah struktur foldernya:



2. Kelas **Buku** harus dibuat sebagai kelas **abstrak** dengan atribut **judul** dan **penulis**, serta memiliki method abstrak **displayInfo()**.

3. Kelas **Buku** harus memiliki dua subclass: **Fiksi** dan **NonFiksi**, di mana masing-masing subclass mengimplementasikan method **displayInfo()** dengan cara yang berbeda.
4. Buatlah interface **Peminjaman** yang memiliki dua method: **pinjamBuku()** dan **kembalikanBuku()**. Kelas **Anggota** harus mengimplementasikan interface ini untuk mencetak keterangan **peminjaman** atau **pengembalian**.
5. Dalam kelas **Anggota**, buatlah method **pinjamBuku()** yang memiliki dua versi, satu menerima parameter berupa **judul buku**, dan satu lagi menerima parameter berupa **judul** dan **durasi peminjaman**. Kemudian buat 2 atribut yaitu:
  - o **String** : nama
  - o **String** : idAnggota
6. Contoh output yang diharapkan:

```
Buku Non-Fiksi: Madilog oleh Tan Malaka (Bidang: Sejarah & Ilmu Pengetahuan )
Buku Fiksi: Hainuwele: Sang Putri Kelapa oleh Lilis Hu (Genre: Dongeng)

Anggota: Wahyu Andika (ID: B075)
Anggota: Ega Faiz (ID: A047)

Wahyu Andika meminjam buku berjudul: Madilog
Ega Faiz meminjam buku "Hainuwele: Sang Putri Kelapa" selama 7 hari.

Wahyu Andika mengembalikan buku berjudul: Madilog
Ega Faiz mengembalikan buku berjudul: Hainuwele: Sang Putri Kelapa

Process finished with exit code 0
```

7. Catatan:
  - o **idAnggota** merupakan **kelas kalian** dan **3-digit nim terakhir** kalian dan teman kalian
  - o **Nama** menggunakan **nama kalian** dan **teman kalian**
  - o **Bagi yang tidak mengumpulkan codelab pada saat praktikum, kumpulkan di Ilab dalam bentuk laporan (pdf).** Laporan harus menjelaskan step by step penggerjaan codelab. Cantumkan link repository kalian yang sudah kalian list di Spreadsheet modul 1 kemarin. Contoh laporan dapat dilihat [disini](#).

## **TUGAS**

Lanjutkan pengembangan program sistem login dari Modul 3. Pada tugas kali ini, Kita akan menerapkan konsep **Package**, **Abstraction (Abstract Class & Method)**, **Interface**, dan **Polymorphism** untuk mengorganisasi kode dengan lebih baik dan mendefinisikan menu serta aksi yang berbeda untuk peran Admin dan Mahasiswa setelah login.

### **Struktur Program & Modifikasi:**

#### **1. Package Organization:**

- Buat package **com.praktikum.users** dan pindahkan kelas **User**, **Admin**, **Mahasiswa** ke dalamnya.
- Buat package **com.praktikum.actions** untuk menampung interface-interface yang akan dibuat.
- Buat package **com.praktikum.main** dan pindahkan kelas **LoginSystem** (atau kelas utama Anda) ke dalamnya.
- Perbarui deklarasi package di awal setiap file dan tambahkan pernyataan **import** yang diperlukan di mana pun kelas/interface dari package lain digunakan (terutama di **LoginSystem** dan mungkin di dalam kelas **Admin/Mahasiswa** jika menggunakan **Scanner** dari **java.util**).

#### **2. Abstraction Enhancement (User Class):**

- Pastikan kelas **User** (di package **com.praktikum.users**) dideklarasikan sebagai **abstract class**.
- Pastikan method **login()** di kelas **User** adalah **abstract method**.
- Tambahkan **abstract method** baru di dalam kelas **User**:

```
// Di dalam abstract class User  
// Method untuk menampilkan menu spesifik peran  
abstract void displayAppMenu();
```

### 3. Interface Implementation (Peran & Aksi):

- **Interface untuk Mahasiswa:**

- Buat interface baru bernama **MahasiswaActions** di package **com.praktikum.actions**.
- Definisikan method signature berikut di dalamnya:

```
// Aksi untuk melaporkan barang
void reportItem();
// Aksi untuk melihat daftar barang (implementasi nanti)
void viewReportedItems();
```

- Buat kelas **Mahasiswa** mengimplementasikan (**implements**) interface **MahasiswaActions**.

- **Interface untuk Admin:**

- Buat interface baru bernama **AdminActions** di package **com.praktikum.actions**.
- Definisikan method signature berikut di dalamnya:

```
// Aksi untuk mengelola barang (implementasi nanti)
void manageItems();
// Aksi untuk mengelola data mahasiswa (implementasi nanti)
void manageUsers();
```

- Buat kelas **Admin** mengimplementasikan (**implements**) interface **AdminActions**.

### 4. Implementasi Method Interface:

- **Di kelas Mahasiswa:**

- Implementasikan method **reportItem()**: Gunakan **Scanner** untuk meminta input detail: **Nama Barang, Deskripsi Barang, Lokasi Terakhir/Ditemukan**. Cetak pesan konfirmasi setelah input diterima.

- Implementasikan method **viewReportedItems()**: Untuk Modul 4 ini, cukup cetak pesan placeholder: **System.out.println(">> Fitur Lihat Laporan Belum Tersedia <<");**
  - **Di kelas Admin:**
    - Implementasikan method **manageItems()**: Cetak pesan placeholder: **System.out.println(">> Fitur Kelola Barang Belum Tersedia <<");**
    - Implementasikan method **manageUsers()**: Cetak pesan placeholder: **System.out.println(">> Fitur Kelola Mahasiswa Belum Tersedia <<");**
5. **Implementasi Menu Peran (displayAppMenu):**
- Implementasikan (**override**) method **displayAppMenu()** di kedua subclass:
    - Di kelas Admin:
      - Tampilkan menu: "**1. Kelola Laporan Barang**", "**2. Kelola Data Mahasiswa**", "**0. Logout**".
      - Jika 1: panggil **manageItems()**;
      - Jika 2: panggil **manageUsers()**;
    - Di kelas Mahasiswa:
      - Tampilkan menu: "**1. Laporkan Barang Temuan/Hilang**", "**2. Lihat Daftar Laporan**", "**0. Logout**".
      - Jika 1: panggil **reportItem()**;
      - Jika 2: panggil **viewReportedItems()**;
      - (Pertimbangkan untuk membuat loop sederhana di dalam **displayAppMenu** agar pengguna bisa memilih beberapa aksi sebelum **logout**, atau kembali ke menu setelah satu aksi selesai).
6. **Polymorphism & Main Logic (LoginSystem):**
- Di kelas **LoginSystem** (package **com.praktikum.main**), setelah login berhasil:
    - Simpan objek **user (Admin atau Mahasiswa)** dalam variabel bertipe **User**.

- Panggil method **displayAppMenu()** melalui variabel **User** tersebut. Pemanggilan ini akan mengeksekusi implementasi **displayAppMenu** yang benar (milik **Admin** atau **Mahasiswa**) karena **polymorphism**.

#### Cara Kerja Program (Setelah Modifikasi):

- Program berjalan dari **LoginSystem**, meminta pilihan login.
- Setelah verifikasi berhasil, objek user disimpan sebagai **User**.
- **displayAppMenu()** dipanggil secara **polimorfik**, menampilkan menu yang sesuai.
- **Admin** dapat memilih menu 1 atau 2 (yang akan memanggil method placeholder dari **AdminActions**).
- **Mahasiswa** dapat memilih menu 1 (memanggil **reportItem** yang meminta input barang) atau menu 2 (memanggil method placeholder **viewReportedItems**).
- Opsi **logout** mengakhiri sesi menu untuk user tersebut.

#### Petunjuk Implementasi:

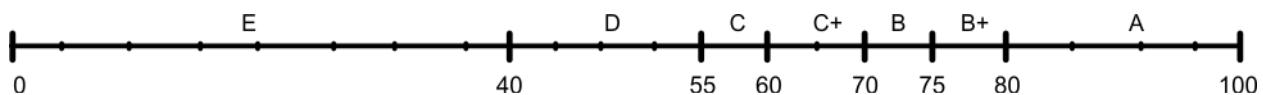
- Pastikan struktur Package benar dan semua **import** sudah sesuai.
- Gunakan **Abstract Class (User)** dan **Abstract Method (login, displayAppMenu)** sebagai fondasi.
- Implementasikan Interface (**MahasiswaActions**, **AdminActions**) untuk mendefinisikan aksi spesifik peran.
- Manfaatkan Polymorphism saat memanggil **displayAppMenu**.
- Fokus implementasi aksi hanya pada **reportItem()** untuk **Mahasiswa** di modul ini; yang lain cukup placeholder.

### PENILAIAN

#### RUBRIK PENILAIAN

Aspek Penilaian	Poin
<b>CODELAB</b>	<b>Total 25%</b>
Kerapian kode	5%
Ketepatan kode & output	10%

Kekreativitasan kode	5%
Kode orisinal (tidak nyontek)	5%
<b>TUGAS</b>	<b>Total 75%</b>
Kerapian kode	5%
Ketepatan kode & output	20%
Kode orisinal (tidak nyontek)	5%
Kemampuan menjelaskan	20%
Menjawab pertanyaan	25%
<b>TOTAL</b>	<b>100%</b>

**SKALA PENILAIAN**

**A** = (81 - 100) → Sepuh

**B+** = (75 - 80) → Sangat baik

**B** = (70 - 74) → Baik

**C+** = (60 - 69) → Cukup baik

**C** = (55 - 59) → Cukup

**D** = (41 - 54) → Kurang

**E** = (0 - 40) → Bro really...

## SUMMARY AKHIR MODUL

Gimana Modul 4 nya? Lumayan *mikir* keras ya? Di modul kali ini, kita udah bongkar konsep-konsep penting PBO seperti **Polymorphism**, **Abstraction** (lewat Abstract Class dan Method), **Interface**, dan juga cara merapikan kode pakai **Package**. Kerasa banget kan, gimana konsep-konsep ini bantu kita bikin struktur program yang lebih logis dan fleksibel, terutama untuk aplikasi yang mulai kompleks.

Pemahaman tentang abstract class dan interface ini memang butuh latihan dan imajinasi. Kapan pakai abstract class? Kapan pakai interface? Kuncinya ada di eksperimen dan coba-coba langsung. Jangan cuma dibaca materinya, tapi *oprek* kodennya, lihat apa yang terjadi kalau diubah-ubah. Kemampuan berpikir abstrak kalian diuji dan diasah di sini.

Seperti biasa, jangan lupa, belajar itu proses panjang. Kalau ada yang *error* atau konsepnya belum 100% nyantol, santai aja, itu wajar banget! Justru dari situ kita belajar paling banyak. Nilai modul kemarin mungkin belum maksimal? Masih ada kesempatan di modul berikutnya dan di kelas teori. Yang penting jangan berhenti mencoba dan bertanya.

Di modul selanjutnya (Modul 5), kita akan masuk ke cara mengelola data yang lebih dinamis menggunakan **Collections** seperti **ArrayList** dan juga cara menangani error biar program kita nggak gampang *crash* dengan **Exceptions**. Jadi, apa yang udah kalian bangun di tugas modul ini, bakal kita kembangkan lagi biar bisa nyimpen banyak data laporan dan lebih tangguh.