

VERSION 2.0

13 Maret 2024



PEMROGRAMAN BERORIENTASI OBJEK

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

DISUSUN OLEH:

TAUFIQ RAMADHAN

SUTRISNO ADIT PRATAMA

DIAUDIT OLEH:

Ir. Galih Wasis Wicaksono, S.Kom, M.Cs.

PRESENTED BY: TIM LAB. IT

UNIVERSITAS MUHAMMADIYAH MALANG

PEMROGRAMAN BERORIENTASI OBJEK

TUJUAN

1. Mahasiswa dapat memahami multi file dengan package.
2. Mahasiswa dapat memahami konsep polymorphism.
3. Mahasiswa dapat memahami konsep abstraction.
4. Mahasiswa dapat memahami method overloading.
5. Mahasiswa dapat memahami abstract class & abstract method.
6. Mahasiswa dapat memahami interface.

TARGET MODUL

1. Mahasiswa dapat membuat dan mengimplementasikan multi file dengan package.
2. Mahasiswa dapat membuat dan mengimplementasikan polymorphism.
3. Mahasiswa dapat membuat dan mengimplementasikan overloading.
4. Mahasiswa dapat membuat dan mengimplementasikan abstract class dan abstract method.
5. Mahasiswa dapat membuat dan mengimplementasikan interface.
6. Mahasiswa dapat membuat dan mengimplementasikan relasi antar kelas.

PERSIAPAN

1. Java Development Kit.
2. Text Editor / IDE (Visual Studio Code, Netbeans, IntelliJ IDEA, atau yang lainnya).

KEYWORDS

- Package
- Polymorphism
- Overloading
- Interface
- Abstract class

- Abstract method
- Concrete class
- Concrete method
- Class relation

TEORI

- **Package**

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.

- Built-in Package (package bawaan java)

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:

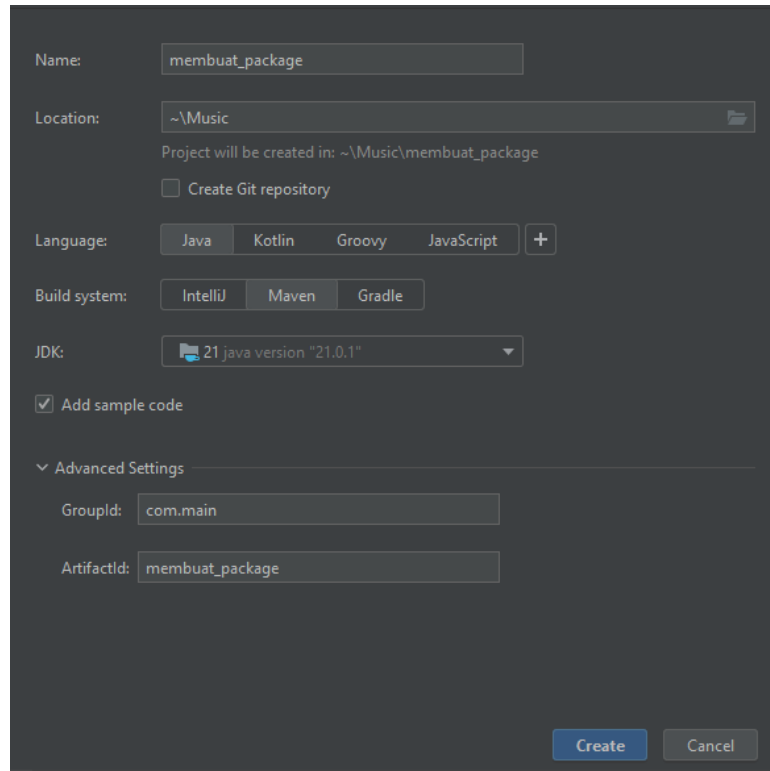
```
import java.util.Scanner;
```

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

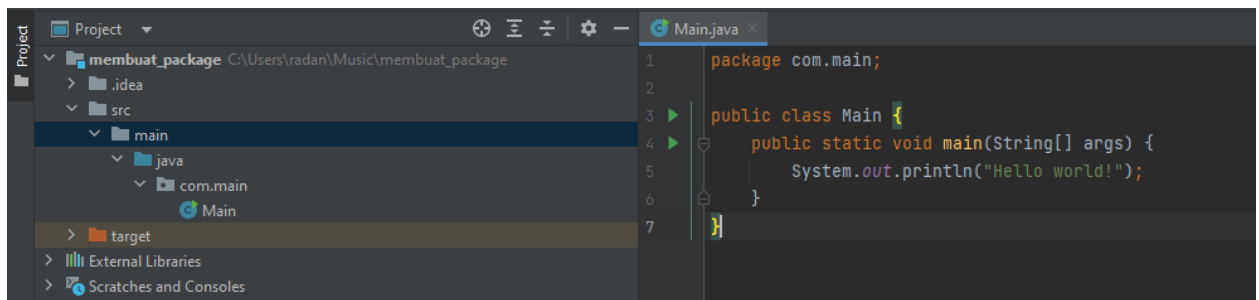
- User-defined package (package buatan kita sendiri)

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

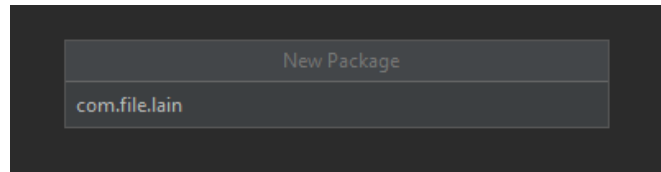
Silahkan buka IntelliJ IDEA dan buat proyek baru nama dan location directory bebas sesuai dengan kebutuhan kalian. Klik bagian Advanced Settings, di sana terdapat GroupId yang dimana ini adalah package default yang akan kita buat yaitu **org.example**. Kalau kita ingin mengubah nama package langsung saja kita ubah dan perlu diperhatikan untuk pembuatan package harus menggunakan tanda titik “.” untuk pemisah, tidak bisa menggunakan spasi. Contoh di sini kita ubah menjadi **com.main**.



Setelah itu klik create, ketika kita lihat pada tab bagian kiri, kita akan melihat seperti ini:



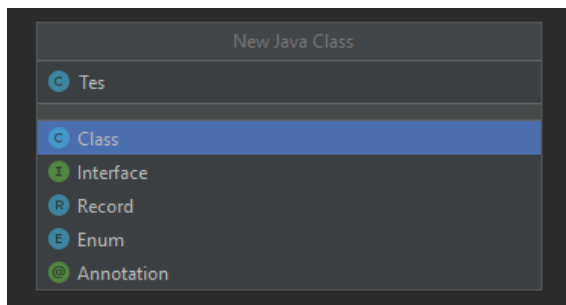
Gambar di atas menunjukkan bahwa file Main.java tersimpan di package **com.main**, hal itu bisa terlihat di tab kiri bahwasannya terdapat folder yaitu com, main, setelah itu baru ada file Main.java. Untuk menambah package baru bisa klik kanan pada file java yang berwarna biru di tab kiri, setelah itu pilih New lalu pilih package. Contoh di sini kita akan membuat sebuah package **com.file.lain**, maka tinggal kita ketik saja di dalam kota form *Name*, setelah itu tekan Enter.



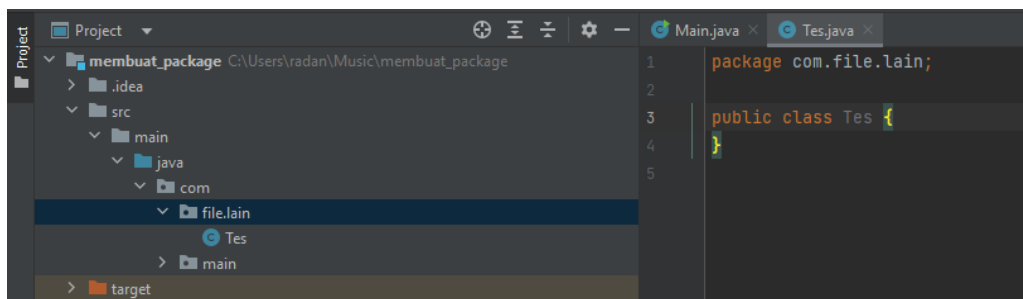
Maka di bagian kiri akan muncul package baru seperti ini:



Untuk menambah file baru di dalam package **file.lain**, klik kanan pada package yang bersangkutan lalu pilih New -> Java class, setelah itu input nama file yang akan dibuat contoh kita buat class Tes, lalu tekan enter:



Maka akan terbentuk sebuah file baru seperti ini:



Untuk cara pakai class Tes yang sudah kita buat di dalam package **file.lain**, kita coba untuk membuat sebuah method sederhana di dalam class Tes. Contohnya kita buat sebuah method cekInfo() dengan return type String yang berisi seperti ini:

```
package com.file.lain;

public class Tes {
    public String cekInfor() {
        return "Method ini berasal dari package file.lain dengan class Tes";
    }
}
```

Untuk cara memakainya bisa harus import terlebih dahulu class Tes di dalam class Main seperti ini:

```
1 package com.main;
2
3 import com.file.lain.Tes;
4
5 public class Main {
6     public static void main(String[] args) {
7         System.out.println("Hello world!");
8     }
9 }
```

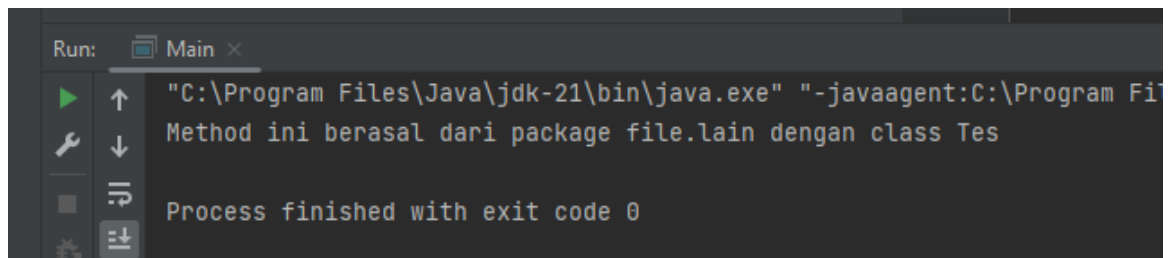
Lalu tinggal kita buat sebuah object dari class Tes seperti berikut:

```
package com.main;

import com.file.lain.Tes;

public class Main {
    public static void main(String[] args) {
        Tes cobaPackage = new Tes();
        System.out.println(cobaPackage.cekInfor());
    }
}
```

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



```
Run: Main x
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Fi
Method ini berasal dari package file.lain dengan class Tes
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**

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 umpakan 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.

Terdapat beberapa tipe Polymorphism yaitu:

1) Ad Hoc Polymorphism (overloading method atau static polymorphism)

Polymorphism adalah teknik polymorphism dimana suatu method yang dapat diaplikasikan dengan berbagai argument atau parameter yang berbeda. Konsep ini juga disebut dengan overloading method atau operator overloading. Tipe ini memungkinkan sebuah method diimplementasikan berbeda-beda berdasarkan argument atau parameter apa yang terdapat pada method. 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
    }

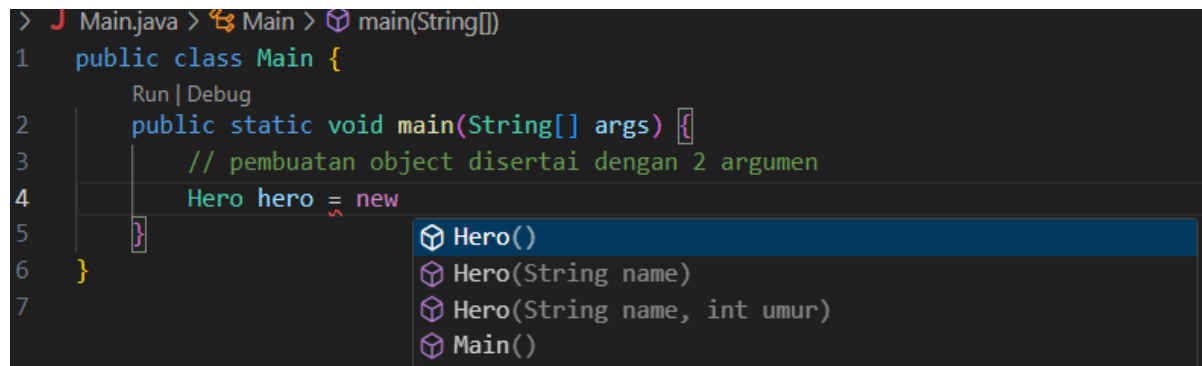
    public void melindungi(){
        System.out.println(name + " melindungi masyarakat");
    }

    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 paramter dan tipenya. Dengan adanya overloading

method seperti di atas maka kita bisa membuat instance object dari class Hero seperti ini tanpa mengalami error:



```

> J Main.java > Main > main(String[])
1 public class Main {
    Run | Debug
2     public static void main(String[] args) {
3         // pembuatan object disertai dengan 2 argumen
4         Hero hero = new
5     }
6 }
7

```

The dropdown menu shows the following options:

- Hero()
- Hero(String name)
- Hero(String name, int umur)
- Main()

```

public class Main {
    public static void main(String[] args) {
        // pembuatan object disertai dengan 2 argumen
        Hero hero = new Hero();
        Hero hero2 = new Hero("Adit");
        Hero hero3 = new Hero("Pratama", 27);
    }
}

```

Terlihat pada kode di atas kita membuat object dengan 3 cara yaitu tanpa paramter, 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) {
        // pembuatan object disertai dengan 2 argumen
        Superman superman = new Superman("Sutrino adit", 33);

        // pemanggilan method yang dilakukan overloading
        superman.attack("monster", 13);
        superman.attack("Giant");
        superman.attack();
    }
}

```

Ketika kita jalankan programnya maka akan output seperti ini:

```

PS C:\Users\radan\Music\pbo modul> & 'C:\
\pbo modul\bin' 'Main'
Superman menyerang monster sebanyak 13
Superman menyerang Giant
Superman menyerang musuh
PS C:\Users\radan\Music\pbo modul>

```

2) 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:

```

PS C:\Users\radan\Music\pbo modul> & 'C:\Users\radan\.jdk\
\pbo modul\bin' 'Main'
Superman melindungi masyarakat bumi dari serangan monster
Hero melindungi masyarakat
PS C:\Users\radan\Music\pbo modul>

```

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) – menkonversi 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) – menkonversi 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 eksplisit 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 eksplisit 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:

```
ppData\Roaming\Code\User\workspaceStorage\c78ac2618751e23d3
Spiderman melindungi masyarakat
Superman melindungi masyarakat bumi dari serangan monster
Deadpool melindungi masyarakat
```

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.

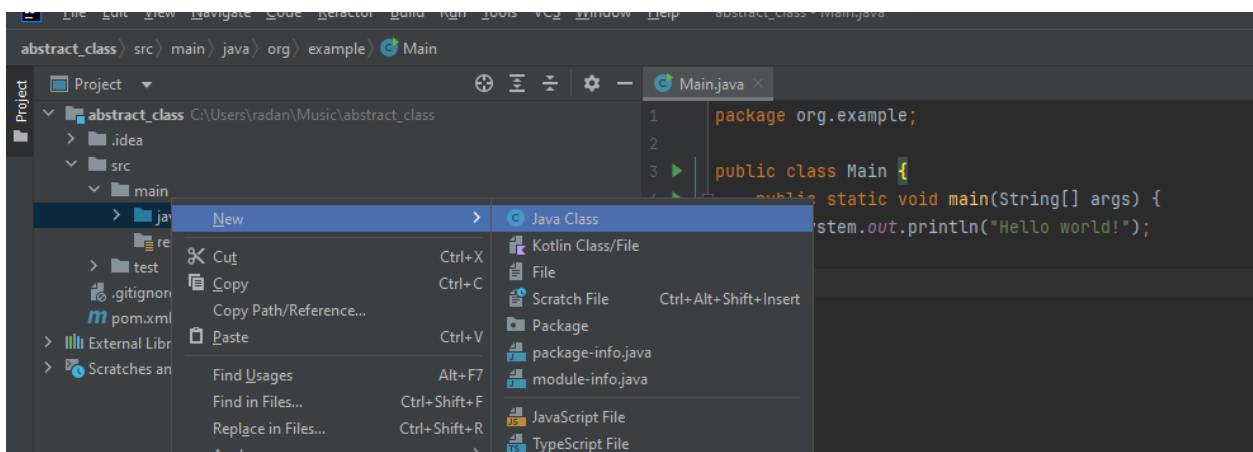
- **Abstract Class**

Abstract class adalah kelas yang dideklarasikan dengan *keyword* *abstract*, dan sifatnya tidak dapat dibuat menjadi suatu objek (tidak dapat diinstansiasi) karena

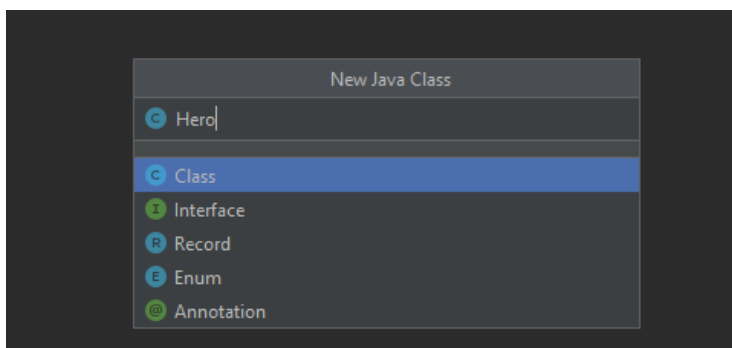
bentuknya memang belum jelas (*abstract*). *Abstract class* biasanya digunakan untuk generalisasi objek yang sangat umum, misalnya Hewan, Kendaraan, *Database*, dan lain sebagainya.

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*.

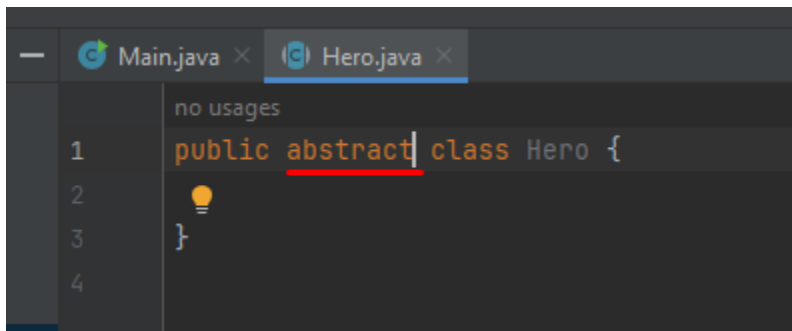
Analogi mengenai class abstract ialah seperti ini, ketika mendengar sebuah kata “hero” tentu yang dipikirkan pertama kali adalah “hero apa yang dimaksud? assasin, marksmen, mage, tank atau apa?” apapun itu, semuanya adalah hero. Kita tahu bahwa semua hero pasti bisa attack, tapi bagaimana kalau “hero 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:

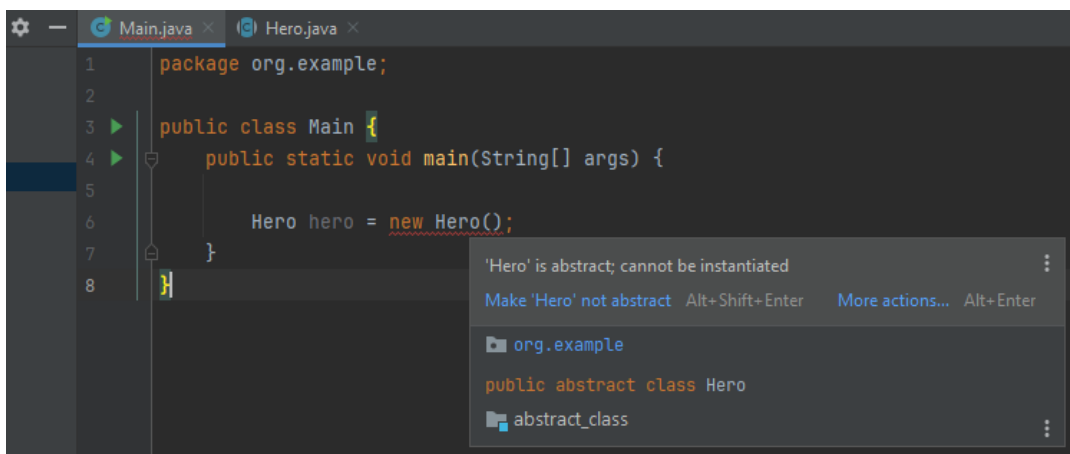


```

1  public abstract class Hero {
2
3  }
4

```

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 abtrak.

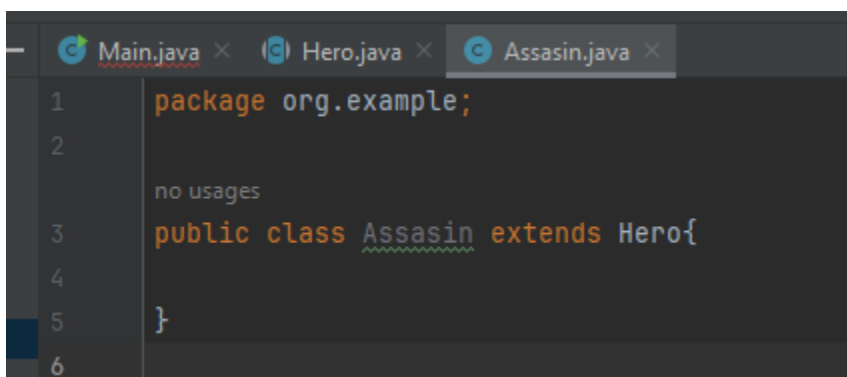


```

1  package org.example;
2
3  public class Main {
4      public static void main(String[] args) {
5
6          Hero hero = new Hero();
7      }
8  }

```

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.

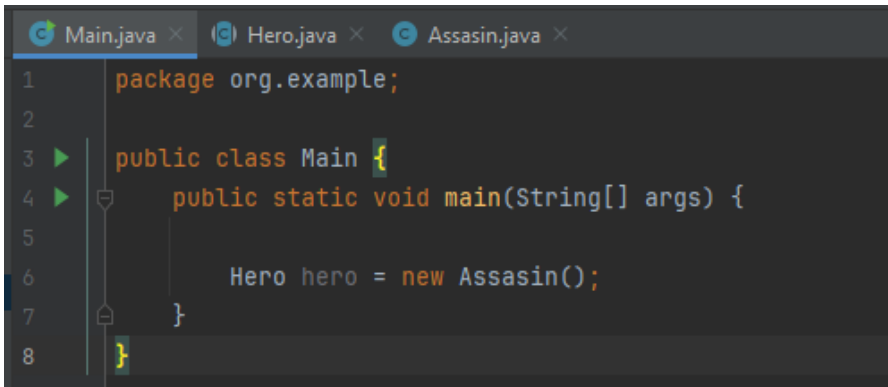


```

1  package org.example;
2
3  public class Assasin extends Hero{
4
5  }
6

```

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



```

1 package org.example;
2
3 public class Main {
4     public static void main(String[] args) {
5
6         Hero hero = new Assassin();
7     }
8 }

```

Sebuah *class abstract* dibuat untuk mengumpulkan sebuah method umum atau sebuah ide suatu *class* berperilaku. Sebagai contoh jika kita membuat sebuah *class* hero Assassin, 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 perilakunya 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*:

```

J Hero.java > Hero
1 public abstract class Hero {
2
3     public abstract void namaMethod();
4
5 }

```

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 error.

→ 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 proyek 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 > J Main.java > Main > main
1 public class Main {
2     public static void main(String[] args) {
3         Hero hero = new Hero();
4     }
5 }
6
```

Cannot instantiate the type Hero Java(16777373)

Hero

View Problem (Alt+F8) No quick fixes available

Terbukti, akan muncul *error* “Cannot instantiate the type Hewan” 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:

```
1
2 public class Tank extends Hero{
3
4 }
5
```

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

The type Tank must implement the inherited abstract method Hero.attack() Java(67109264)

Tank

View Problem (Alt+F8) Quick Fix... (Ctrl+.)

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 VS Code 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.

```
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!

```

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

• 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. | Secara implisit 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 variable dalam interface adalah public static final. |
| Hanya dapat meng- <i>extends</i> satu abstract class lainnya (tidak bisa multiple inheritance). | Dapat meng- <i>implements</i> lebih dari 1 interface. |
| Dapat meng- <i>implements</i> interface lebih | Tidak dapat meng- <i>implements</i> interface |

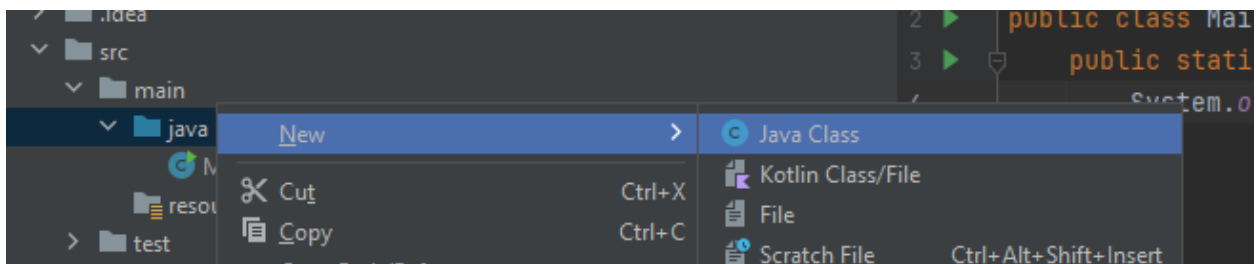
| | |
|-------------------------------------|--------------------------------|
| dari satu interface. | lain. |
| Bisa membuat properti atau variabel | Hanya bisa buat konstanta saja |

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*.

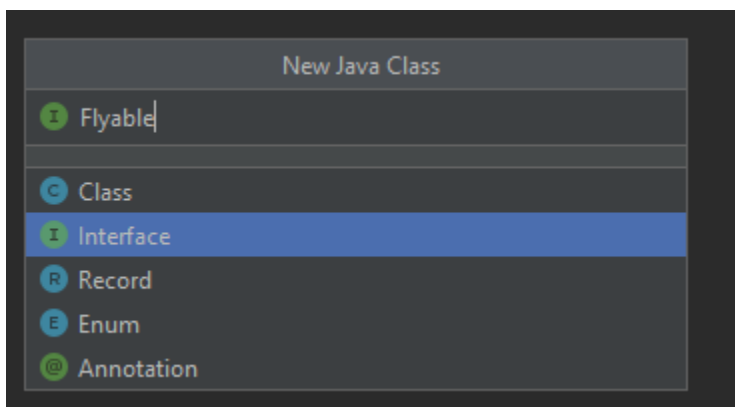
Contoh pendeklarasian *interface*:

```
// 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

```
1 public class Burung implements Flyable{
2
3 }
4
```

The type Burung must implement the inherited abstract method Flyable.terbang() Java(67109264)

Burung

View Problem (Alt+F8) Quick Fix... (Ctrl+.)

File Pesawat.java

```
src > Pesawat.java > Pesawat
1 public class Pesawat implements Flyable{
2
3 }
4
```

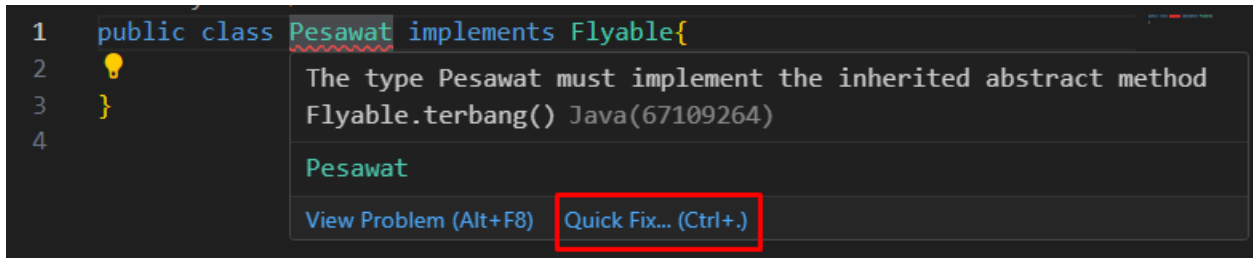
The type Pesawat must implement the inherited abstract method Flyable.terbang() Java(67109264)

Pesawat

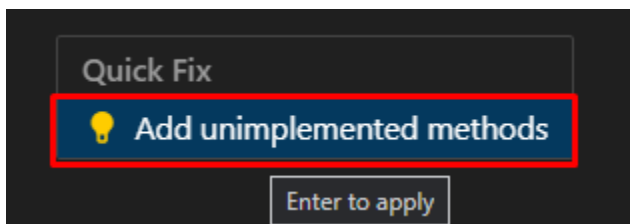
View Problem (Alt+F8) Quick Fix... (Ctrl+.)

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 *Quick fix*.



Setelah itu bisa di klik opsi yang ada:



Maka kode akan berubah tidak error seperti ini:

```
public class Pesawat implements Flyable{
    @Override
    public void terbang() {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException("Unimplemented method
'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) {
        // pembuatan objek dengan cara biasa
        Burung burung = new Burung();
        Pesawat pesawat = new Pesawat();

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

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

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

```

Ketika dijalankan maka akan keluar sebuah output seperti ini:

```

\pbo modul\bin' 'Main'
Burung terbang menggunakan sayap
Burung terbang menggunakan sayap
Pesawat terbang menggunakan mesin
Pesawat terbang menggunakan mesin

```

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));
```

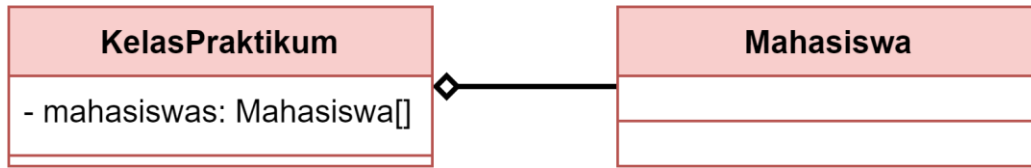
Bahkan sebelum kalian *run* IDE kalian akan menunjukkan kepada kalian bahwa kode tersebut akan error. Jika tetap kalian *run*, *compiler* akan menampilkan error “*Incompatible conditional operand types Tank and Marksman*”. Ini dikarenakan Tank meskipun sebuah Hero tetapi ia bukanlah Marksman.

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("202210370311203", "Sutrisno Adit Pratama");
        KelasPraktikum kelasPraktikum = new KelasPraktikum("PBO C", mahasiswa);



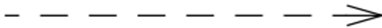



        System.out.println("Kelas praktikum " + kelasPraktikum.name + " memiliki mahasiswa " + kelasPraktikum.mahasiswa.nama);
    }
}
  
```

Output program:

Kelas praktikum PBO C memiliki mahasiswa Sutrisno Adit Pratama

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 proyek Java, Anda sudah mengimplementasikan relasi yang lain. Untuk memahami teori relasi yang lain, silakan Anda berselancar di internet.

CODELAB

Silahkan pull kode yang ada di github berikut di [sini](#). Silahkan perbaiki kode tersebut dengan ketentuan seperti ini:

- Kode pada file Main.java tidak perlu ada yang diubah ataupun diperbaiki
 - Pada file Kendaraan.java silahkan buat method abstract bernama **Stop()** dan **Brake()**
 - Di interface ShootAble dan FlyAble silahkan perbaiki kode interface dan method-nya, seharusnya seperti apa
 - Di dalam class Motor, harap teliti satu-satu method apa yang masih kurang dan harus ada?
 - Pada file Pesawat.java silahkan perbaiki method yang seharusnya dilakukan override dari parent
 - Terakhir di Tank.java, silahkan perbaiki kode bagian apa yang kurang dan menyebabkan error
- Jika sudah selesai dengan semua perbaikan berarti pengerjaan codelab sudah selesai

TUGAS

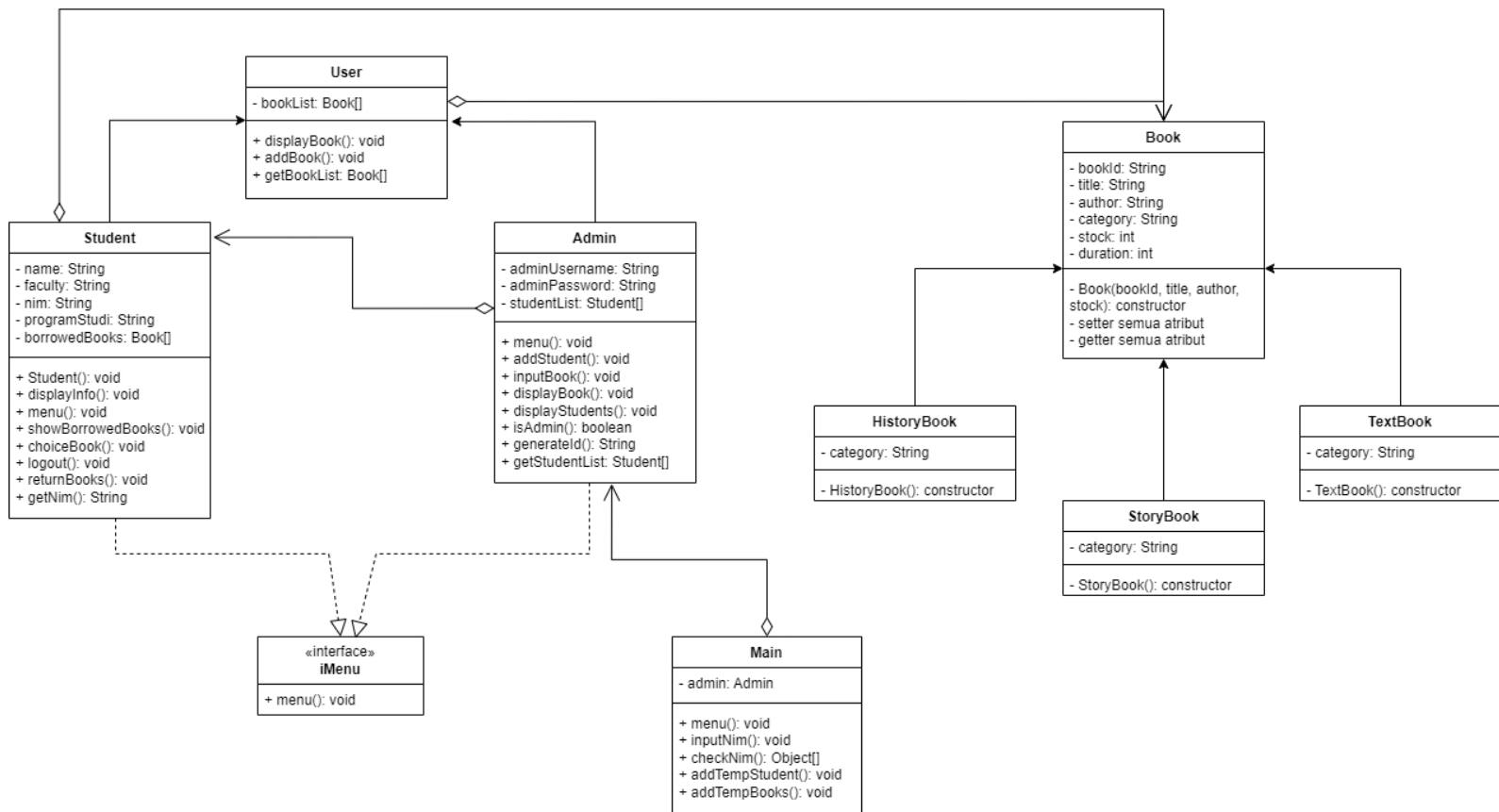
Melanjutkan tugas pada modul 3 sebelumnya, pada tugas modul 4 ini silahkan kembangkan program menjadi dengan ketentuan berikut:

- buat package **com.main** dan taruh file **LibrarySystem.java** ke dalamnya
- buat package **books** yang dimana di dalamnya terdapat class Book, HistoryBook, StoryBook, dan TextBook
- buat package **data** dan di dalamnya terdapat class Admin, Student, dan User
- terakhir buat package **util** dan di dalamnya buat sebuah interface iMenu.java
- Interface **iMenu.java** berisi sebuah method abstract **menu()** dan lakukan implements pada class **Student** dan **Admin**. Sehingga admin dan student memiliki opsi menu di class masing-masing dengan hasil *override* dari interface **iMenu**.
- Silahkan buat sebuah method yang melakukan *overloading* dan terapkan pada code yang sedang dibuat, bebas method apa saja (opsional).
- Jika penyimpanan buku pada kode di tugas modul 2 menggunakan sebuah array multidimensi, harap diganti dengan array satu dimensi dengan cara menggunakan teknik

polymorphism dengan tipe **class Book**, sehingga semua *class HistoryBook*, *StoryBook*, dan *TextBook* bisa disimpan.

- Method **displayBooks()** pada class *Student* diganti menjadi **choiceBook()**, yang di mana nanti method ini tidak hanya digunakan untuk menampilkan buku tetapi juga bisa memilih buku yang ingin dipinjam.

Bentuk diagram class akan menjadi seperti berikut:



Diperbolehkan bagi para mahasiswa praktikum PBO untuk melakukan sebuah improvisasi dari kode program yang sudah didefinisikan di atas, tetapi tidak boleh untuk mengurangi spesifikasi apapun pada ketentuan yang sudah disebutkan. Jika sudah selesai mengerjakan tugas, harap untuk kembali melakukan push ke repository yang sama dengan tugas modul 1-3.

RUBRIK PENILAIAN

| ASPEK PENILAIAN | POIN |
|------------------------|-------------|
| Codelab | 20 |
| Tugas | 30 |
| Tugas opsional | 10 |
| Pemahaman | 40 |
| TOTAL | 100 |

Selamat Mengerjakan
Tetap Semangat