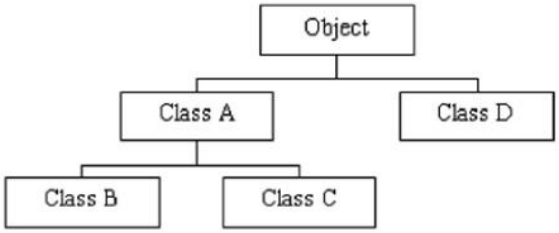
Pertemuan V

PENDEKATAN BERORIENTASI OBYEK

5.1. Pewarisan

Dalam Java, semua *class*, termasuk *class* yang membangun Java API, adalah *subclasses* dari *superclass Object*. Contoh hirarki *class* diperlihatkan pada gambar 5.1 dibawah ini. Beberapa *class* di atas *class* utama dalam hirarki *class* dikenal sebagai *superclass*. Sementara beberapa *class* di bawah *class* pokok dalam hirarki *class* dikenal sebagai *subclass* dari *class* tersebut.



Gambar 5.1. Hirarki Kelas dalam Java

Pewarisan adalah keuntungan besar dalam pemrograman berbasis obyek karena suatu sifat atau metode didefinisikan dalam *superclass*, sifat ini secara otomatis diwariskan dari semua *subclasses*. Jadi kita dapat menuliskan kode metode hanya sekali dan mereka dapat digunakan oleh semua *subclass*. *Subclass* hanya perlu mengimplementasikan perbedaannya sendiri dengan induknya.

5.1.1. Mendefinisikan *Superclass* dan *Subclass*

Untuk memperoleh suatu *class*, kita menggunakan kata kunci **extend**. Untuk mengilustrasikan ini, kita akan membuat contoh *class* induk. Dimisalkan kita mempunyai *class* induk yang dinamakan Person.

```
public class Person{
  protected String name;
  protected String address;

  /**
  * Default constructor
  */
  public Person(){
    System.out.println("Inside Person:Constructor");
    name = "";
    address = "";
  }

  /**
  * Constructor dengan dua parameter
  */
  public Person( String name, String address ){
    this.name = name;
    this.address = address;
  }
```

```
  /**
  * Method accessor
  */
  public String getName(){
    return name;
  }

  public String getAddress(){
    return address;
  }

  public void setName( String name ){
    this.name = name;
  }

  public void setAddress( String add ){
    this.address = add;
  }
}
```

Perhatikan bahwa atribut *name* dan *address* dideklarasikan sebagai protected. Alasannya kita melakukan ini yaitu, kita inginkan atribut-atribut ini untuk bisa diakses oleh *subclasses* dari *superclassess*. Jika kita mendeklarasikannya sebagai *private*, *subclasses* tidak dapat menggunakannya. (Catatan bahwa semua properti dari *superclass* yang dideklarasikan sebagai public, protected dan default dapat diakses oleh *subclasses*-nya).

Sekarang, kita ingin membuat *class* lain bernama Student. Karena Student juga sebagai Person, kita putuskan hanya meng-*extend class* Person, sehingga kita dapat mewariskan semua properti dan metode dari setiap *class* Person yang ada. Untuk melakukan ini, kita tulis kode program berikut ini :

```
public class Student extends Person{
  public Student(){
    System.out.println("Inside Student:Constructor");
    //beberapa kode di sini
  }

  // beberapa kode di sini
}
```

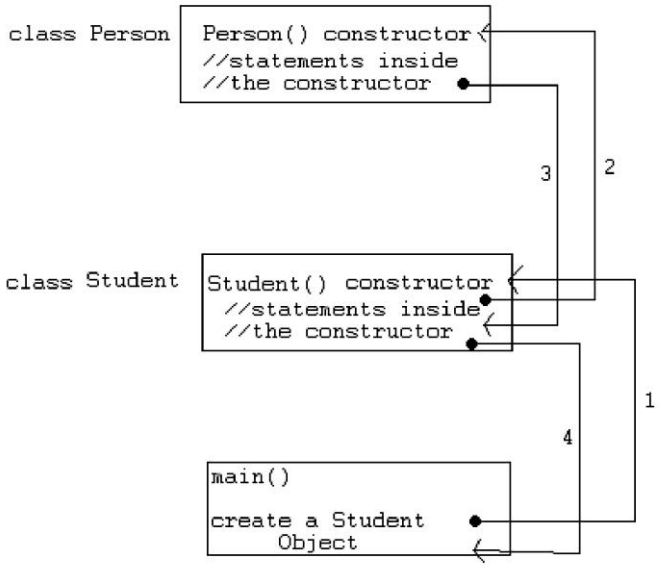
Ketika obyek Student di-*instantiate*, *default constructor* dari *superclass* secara mutlak meminta untuk melakukan inisialisasi yang seharusnya. Setelah itu, pernyataan di dalam *subclass* dieksekusi. Untuk mengilustrasikannya, ketik kode program dibawah ini didalam class Student.

```
public static void main( String[] args ){
  Student anna = new Student();
}
```

Dalam kode ini, kita membuat sebuah obyek dari *class* Student. Keluaran dari program adalah :

```
Inside Person:Constructor
Inside Student:Constructor
```

Ilustrasi alur program ditunjukkan pada gambar 5.2 dibawah ini :



Gambar 5.2. Alur program *Superclass* dan *subclass*

5.1.2. Kata Kunci Super

Subclass juga dapat memanggil *constructor* secara eksplisit dari *superclass* terdekat. Hal ini dilakukan dengan pemanggil *constructor* super. Pemanggil *constructor super* dalam *constructor* dari *subclass* akan menghasilkan eksekusi dari *superclass constructor* yang bersangkutan, berdasar dari argumen sebelumnya.

Sebagai contoh, pada contoh *class* sebelumnya. Person dan Student, kita tunjukkan contoh dari pemanggil *constructor super*. Diberikan kode berikut untuk Student :

```
public Student(){
  super( "SomeName", "SomeAddress" );
  System.out.println("Inside Student:Constructor");
}
```

Kode ini memanggil constructor kedua dari superclass terdekat (yaitu Person) dan mengeksekusinya. Contoh kode lain ditunjukkan sebagai berikut :

```
public Student(){
  super();
  System.out.println("Inside Student:Constructor");
}
```

Kode ini memanggil *default constructor* dari *superclass* terdekat (yaitu Person) dan mengeksekusinya.

Ada beberapa hal yang harus diingat ketika menggunakan pemanggil constuktor super:

a. Pemanggil super() harus dijadikan pernyataan pertama dalam constructor.
b. Pemanggil super() hanya dapat digunakan dalam definisi constructor.
c. Termasuk constructor this() dan pemanggil super() tidak boleh terjadi dalam constructor yang sama.

Pemakaian lain dari super adalah untuk menunjuk anggota dari superclass (seperti reference this). Sebagai contoh :

```
public Student(){
  super.name = "somename";
  super.address = "some address";
}
```

5.1.3. Overriding Method

Untuk beberapa pertimbangan, terkadang *class* asal perlu mempunyai implementasi berbeda dari metode yang khusus dari *superclass* tersebut. Oleh karena itulah, metode *overriding* digunakan. *Subclass* dapat mengesampingkan metode yang didefinisikan dalam *superclass* dengan menyediakan implementasi baru dari metode tersebut.

Misalnya kita mempunyai implementasi berikut untuk metode *getName* dalam *superclass* Person :

```
public class Person{
  :
  :
  public String getName(){
    System.out.println("Parent: getName");
    return name;
  }
  :
}
```

Untuk override, metode getName dalam subclass Student, kita tulis :

```
public class Student extends Person{
  :
  :
  public String getName(){
    System.out.println("Student: getName");
    return name;
  }
  :
}
```

Jadi, ketika kita meminta metode getName dari obyek class Student, metode *overridde* akan dipanggil, keluarannya akan menjadi :

```
Student: getName
```

5.1.4. Method final dan class final

Dalam Java, juga memungkinkan untuk mendeklarasikan class-class yang tidak lama menjadi *subclass*. *Class* ini dinamakan class final. Untuk mendeklarasikan *class* untuk menjadi *final* kita hanya menambahkan kata kunci final dalam deklarasi *class*. Sebagai contoh, jika kita ingin *class* Person untuk dideklarasikan final, kita tulis :

```
public final class Person {
  //area kode
}
```

Beberapa *class* dalam Java API dideklarasikan secara final untuk memastikan sifatnya tidak dapat di-*override*. Contoh-contoh dari class ini adalah *Integer*, *Double*, dan *String*.

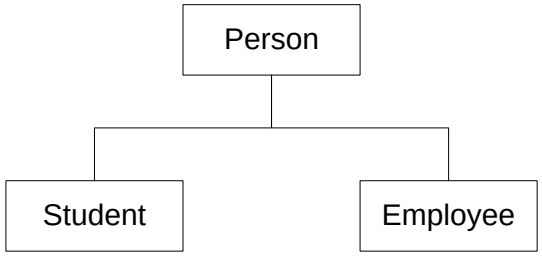
Ini memungkinkan dalam Java membuat metode yang tidak dapat di-*override.* Metode ini dapat kita panggil method final. Untuk mendeklarasikan metode untuk menjadi final, kita tambahkan kata kunci final ke dalam deklarasi metode. Contohnya, jika kita ingin metode getName dalam *class* Person untuk dideklarasikan final, kita tulis :

```
public final String getName(){
  return name;
}
```

Metode static juga secara otomatis final. Ini artinya Anda tidak dapat membuatnya override.

5.2. Polimorfisme

Sekarang, *class* induk Person dan *subclass* Student dari contoh sebelumnya, kita tambahkan *subclass* lain dari Person yaitu Employee. Di bawah ini adalah hierarkinya :



Gambar 5.3. Hirarki dari *class* induk Person

Dalam Java, kita dapat membuat referensi yang merupakan tipe dari *superclass* ke sebuah obyek dari *subclass* tersebut. Sebagai contohnya :

```
public static main( String[] args ){
  Person ref;

  Student studentObject = new Student();
  Employee employeeObject = new Employee();
  ref = studentObject; //Person menunjuk kepada object Student

  //beberapa kode di sini
}
```

Sekarang dimisalkan kita punya metode getName dalam *superclass* Person kita, dan kita override metode ini dalam kedua *subclasses* Student dan Employee:

```
public class Person{
  public String getName(){
    System.out.println("Person Name: " + name);
    return name;
  }
}
```

```
public class Student extends Person{
  public String getName(){
    System.out.println("Student Name: " + name);
    return name;
  }
}

public class Employee extends Person{
  public String getName(){
    System.out.println("Employee Name: " + name);
    return name;
  }
}
```

Kembali ke metode utama kita, ketika kita mencoba memanggil metode getName dari reference Person ref, metode getName dari obyek Student akan dipanggil. Sekarang, jika kita berikan ref ke obyek Employee, metode getName dari Employee akan dipanggil.

```
public static main( String[] args ){
  Person ref;

  Student studentObject = new Student();
  Employee employeeObject = new Employee();

  ref = studentObject; //Person menunjuk kepada object Student
  String temp = ref.getName(); //getName dari Student class dipanggil
  System.out.println( temp );

  ref = employeeObject; //Person menunjuk kepada object Employee
  String temp = ref.getName(); //getName dari Employee class dipanggil
  System.out.println( temp );
}
```

Kemampuan dari *reference* untuk mengubah sifat menurut obyek apa yang dijadikan acuan dinamakan polimorfisme. Polimorfisme menyediakan *multiobject* dari *subclasses* yang berbeda untuk diperlakukan sebagai obyek dari *superclass* tunggal, secara otomatis menunjuk metode yang tepat untuk menggunakannya ke *particular* obyek berdasar *subclass* yang termasuk di dalamnya.

Contoh lain yang menunjukkan properti polimorfisme adalah ketika kita mencoba melalui reference ke metode. Misalkan kita punya metode static printInformation yang mengakibatkan obyek Person sebagai reference, kita dapat me-reference dari tipe Employee dan tipe Student ke metode ini selama itu masih subclass dari class Person.

```
public static main( String[] args ){
  Student studentObject = new Student();
  Employee employeeObject = new Employee();
  printInformation( studentObject );
  printInformation( employeeObject );
}

public static printInformation( Person p ){
  //beberapa kode di sini
}
```

5.3. *Abstract Class*

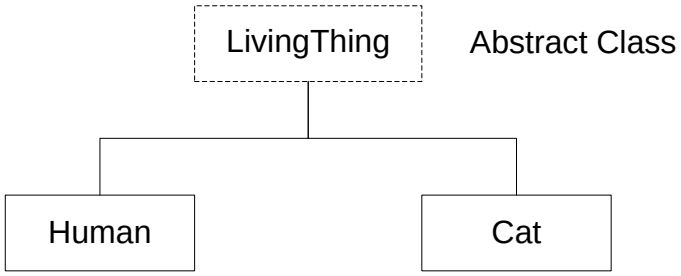
Abstraksi merupakan cara paling dasar dalam menangani kompleksitas. Abtraksi adalah kemampuan manusia untuk mengenali keserupaan antara obyek-obyek, situasi-situasi, atau proses-proses yang berbeda didunia nyata dan keputusan untuk berkonsentrasi pada keserupaan-keserupaan dengan mengabaikan perbedaan-perbedaan kecil yang ada.

Misalnya, manusia dapat menghitung kecepatan dan waktu yang ditempuh saat benda jatuh dari ketinggian dengan mengasumsikan benda yang jatuh sebagai titik masa dengan persamaan kecepatan yang dipercepat dan mengabaikan benda yang jatuh adalah besi, kayu, batu dan lain-lain, dan mengabaikan hambatan.

Abstract Class adalah *class* yang diimplementasikan secara parsial dengan tujuan untuk kenyamanan perancangan. Kelas-kelas abstrak disusun dari satu metode abstrak atau lebih dimana metode-metode dideklarasikan tapi tanpa badan (tidak diimplementasikan), tetapi harus di-*overridden* oleh *subclasses*-nya.

Misalnya kita ingin membuat *superclass* yang mempunyai metode tertentu yang berisi implementasi, dan juga beberapa metode yang akan di-*overridden* oleh *subclasses*-nya.

Sebagai contoh, kita akan membuat *superclass* bernama *LivingThing*, *class* ini mempunyai metode tertentu seperti *breath*, *eat*, dan *walk*. Akan tetapi, ada beberapa metode di dalam *superclass* yang sifatnya tidak dapat digeneralisasi. Kita ambil contoh, metode *walk*, tidak semua kehidupan berjalan (*walk*) dalam cara yang sama. Misalnya, manusia berjalan dengan dua kaki, sedangkan kucing berjalan dengan empat kaki. Akan tetapi, beberapa ciri umum dalam kehidupan sudah biasa, itulah kenapa kita inginkan membuat *superclass*.



Gambar 5.4. *Class Abstract*

Kita dapat membuat *superclass* yang mempunyai beberapa metode dengan implementasi, sedangkan yang lain tidak. *Class* jenis ini yang disebut dengan *abstract class*.

Sebuah *abstract class* adalah *class* yang tidak dapat di-*instantiate*. Seringkali muncul diatas hirarki *class* pemrograman berbasis obyek, dan mendefinisikan keseluruhan aksi yang mungkin pada obyek dari seluruh *subclasses* dalam *class*.

Metode dalam *abstract class* yang tidak mempunyai implementasi dinamakan *method abstract*. Untuk membuat metode abstrak, tinggal menulis deklarasi metode tanpa tubuh dan menggunakan kata kunci *abstract*. Contohnya :

```
public abstract void MethodName();
```

Sekarang kita buat *abstract class* sebagai berikut :

```
abstract class LivingThing
{
  public void breath(){
    System.out.println("Bernafas melalui hidung.");
  }

  public void eat(){
    System.out.println("Makan melalui mulut.");
  }

  /**
  * abstract method walk
  * Kita ingin method ini di-overridden oleh subclasses
  */
  public abstract void walk();
}
```

Ketika *class* meng-*extend abstract class LivingThing*, dibutuhkan *override* untuk metode *abstract walk*(), Contohnya :

```
class Human extends LivingThing
{
  public void walk(){
    System.out.println("Berjalan dengan 2 kaki.");
  }
}
```

Jika *class Human* tidak meng-*overridde* metode *walk*, kita akan menemui pesan error berikut ini :

```
 MakhlukHidup.java:18: Human is not abstract and does not override
abstract method walk() in LivingThing
 class Human extends LivingThing
 ^
 1 error
```

Gunakan *abstract class* untuk mendefinisikan secara luas sifat-sifat dari *class* tertinggi pada hirarki pemrograman berbasis obyek, dan gunakan *subclasses*-nya untuk menyediakan rincian dari *abstract class*.

5.4. *Interface*

Interface adalah prototipe untuk kelas dan berguna ditinjau dari prespektif rancangan logis. Deskripsi ini hampir serupa dengan kelas abstrak, tetapi tidak sama. Kelas abstrak adalah kelas yang diimplementasikan secara parsial. *Interface* adalah kelas abstrak yang sepenuhnya tidak diimplementasikan, yang berarti tidak ada metode yang diimplementasikan dan data anggota di *interface* dapat dipastikan final statis yang berarti konstan murni.

Interface adalah jenis khusus dari blok yang hanya berisi metode *signature* (atau *constant*). *Interface* mendefinisikan sebuah (*signature*) dari sebuah kumpulan metode tanpa tubuh. *Interface* mendefinisikan sebuah cara standar dan umum dalam menetapkan sifat-sifat dari *class-class*. Mereka menyediakan *class-class*, tanpa

memperhatikan lokasinya dalam hirarki *class*, untuk mengimplementasikan sifat-sifat yang umum. Dengan catatan bahwa *interface-interface* juga menunjukkan *polimorfisme*, dikarenakan program dapat memanggil metode *interface* dan versi yang tepat dari metode yang akan dieksekusi tergantung dari tipe obyek yang melewati pemanggil metode *interface*.

5.4.1. Mengapa Kita Memakai Interface ?

Kita akan menggunakan *interface* jika kita ingin *class* yang tidak berhubungan mengimplementasikan metode yang sama. Melalui *interface-interface*, kita dapat menangkap kemiripan diantara *class* yang tidak berhubungan tanpa membuatnya seolah-olah *class* yang berhubungan.

Mari kita ambil contoh *class Line* dimana berisi metode yang menghitung panjang dari garis dan membandingkan obyek *Line* ke obyek dari *class* yang sama. Sekarang, misalkan kita punya *class* yang lain yaitu *MyInteger* dimana berisi metode yang membandingkan obyek *MyInteger* ke obyek dari *class* yang sama. Seperti yang kita lihat disini, kedua *class-class* mempunyai metode yang mirip dimana membandingkan mereka dari obyek lain dalam tipe yang sama, tetapi mereka tidak berhubungan sama sekali. Supaya dapat menjalankan cara untuk memastikan bahwa dua *class-class* ini mengimplementasikan beberapa metode dengan tanda yang sama, kita dapat menggunakan sebuah interface untuk hal ini. Kita dapat membuat sebuah class interface, katakanlah interface Relation dimana mempunyai deklarasi metode pembanding. Relasi interface dapat dideklarasikan sebagai :

```
public interface Relation
{
  public boolean isGreater( Object a, Object b);
  public boolean isLess( Object a, Object b);
  public boolean isEqual( Object a, Object b);
}
```

Alasan lain dalam menggunakan *interface* pemrograman obyek adalah untuk menyatakan sebuah *interface* pemrograman obyek tanpa menyatakan *classnya.* Seperti yang dapat kita lihat nanti dalam bagian *Interface vs class*, kita dapat benar-benar menggunakan interface sebagai tipe data.

Pada akhirnya, kita perlu menggunakan *interface* untuk pewarisan model jamak dimana menyediakan class untuk mempunyai lebih dari satu *superclass*. Pewarisan jamak tidak ditunjukkan di Java, tetapi ditunjukkan di bahasa berorientasi obyek lain seperti C++.

5.4.2. *Interface vs. Class Abstract*

Berikut ini adalah perbedaan utama antara sebuah *interface* dan sebuah *class abstract*: metode *interface* tidak punya tubuh, sebuah *interface* hanya dapat mendefinisikan konstanta dan *interface* tidak langsung mewariskan hubungan dengan *class* istimewa lainnya, mereka didefinisikan secara *independent*.

5.4.3. *Interface vs. Class*

Satu ciri umum dari sebuah interface dan class adalah pada tipe mereka berdua. Ini artinya bahwa sebuah interface dapat digunakan dalam tempat-tempat dimana sebuah *class* dapat digunakan. Sebagai contoh, diberikan *class Person* dan *interface PersonInterface*, berikut deklarasi yang benar :

```
PersonInterface pi = new Person();
Person pc = new Person();
```

Bagaimanapun, kita tidak dapat membuat *instance* dari sebuah *interface*. Contohnya:

```
PersonInterface pi = new PersonInterface(); //COMPILE ERROR!!!
```

Ciri umum lain adalah baik *interface* maupun *class* dapat mendefinisikan metode. Bagaimanapun, sebuah *interface* tidak punya sebuah kode implementasi sedangkan *class* memiliki salah satunya.

5.4.4. Membuat *Interface*

Untuk membuat *interface*, kita tulis :

```
public interface [InterfaceName]
{
  //beberapa method tanpa isi
}
```

Sebagai contoh, mari kita membuat sebuah *interface* yang mendefinisikan hubungan antara dua obyek menurut urutan asli dari obyek.

```
public interface Relation
{
  public boolean isGreater( Object a, Object b);
  public boolean isLess( Object a, Object b);
  public boolean isEqual( Object a, Object b);
}
```

Sekarang, penggunaan *interface*, kita gunakan kata kunci *implements*. Contohnya :

```
/**
* Class ini mendefinisikan segmen garis
*/
public class Line implements Relation
{
  private double x1;
  private double x2;
  private double y1;
  private double y2;

  public Line(double x1, double x2, double y1, double y2){
    this.x1 = x1;
    this.x2 = x2;
    this.y1 = y1;
    this.y2 = y2;
  }

  public double getLength(){
    double length = Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)* (y2-y1));
    return length;
  }

  public boolean isGreater( Object a, Object b){
    double aLen = ((Line)a).getLength();
```

```
    double bLen = ((Line)b).getLength();
    return (aLen > bLen);
  }

  public boolean isLess( Object a, Object b){
    double aLen = ((Line)a).getLength();
    double bLen = ((Line)b).getLength();
    return (aLen < bLen);
  }

  public boolean isEqual( Object a, Object b){
    double aLen = ((Line)a).getLength();
    double bLen = ((Line)b).getLength();
    return (aLen == bLen);
  }
}
```

Ketika mengimplementasikan sebuah *interface* pada sebuah *class*, selalu pastikan bahwa semua metode dari *interface* telah diimplementasikan. Jika tidak, maka akan ada kesalahan seperti ini :

```
Line.java:4: Line is not abstract and does not override abstract
method isGreater(java.lang.Object,java.lang.Object) in Relation
public class Line implements Relation
^
1 error
```

Gunakan *interface* untuk mendefinisikan metode standar yang sama dalam *class-class* berbeda yang memungkinkan. Sekali kita telah membuat kumpulan definisi metode standar, kita dapat menulis metode tunggal untuk memanipulasi semua *class-class* yang mengimplementasikan *interface.*

5.4.5. Hubungan dari *Interface* ke *Class*

Seperti yang telah kita lihat dalam bagian sebelumnya, *class* dapat mengimplementasikan sebuah *interface* selama kode implementasi untuk semua metode yang didefinisikan dalam *interface* tersedia.

Hal lain yang perlu dicatat tentang hubungan antara *interface* ke *class-class* yaitu, *class* hanya dapat meng-*extend* satu *superclass*, tetapi dapat mengimplementasikan banyak *interface*. Sebuah contoh dari sebuah *class* yang mengimplementasikan *interface* adalah :

```
public class Person implements
PersonInterface,LivingThing,WhateverInterface {
    //beberapa kode di sini
}
```

Contoh lain dari *class* yang meng-*extend* satu *superclass* dan mengimplementasikan sebuah *interface* adalah :

```
public class ComputerScienceStudent extends Student implements
PersonInterface, LivingThing {
    //beberapa kode di sini
}
```

Catatan bahwa sebuah *interface* bukan bagian dari hirarki pewarisan *class*. *Class* yang tidak berhubungan dapat mengimplementasikan *interface* yang sama.

5.4.6. Pewarisan Antar *Interface*

Interface bukan bagian dari hirarki *class*. Bagaimanapun, *interface* dapat mempunyai hubungan pewarisan antara mereka sendiri. Contohnya, misal kita punya dua interface *StudentInterface* dan *PersonInterface*. Jika *StudentInterface* meng-*extend PersonInterface*, maka ia akan mewariskan semua deklarasi metode dalam *PersonInterface*.

```
public interface PersonInterface {
  . . .
}

public interface StudentInterface extends PersonInterface {
  . . .
}
```

5.5. Contoh Dalam Program

5.5.1. Abstract Class

```
abstract class LivingThing
{
  public void breath(){
    System.out.println("Bernafas melalui hidung.");
  }

  public void eat(){
    System.out.println("Makan melalui mulut.");
  }

  /**
  * abstract method walk
  * Kita ingin method ini di-overridden oleh subclasses
  */
  public abstract void walk();
}

class Human extends LivingThing
{
  /** Mengimplementasikan metode abstrak */
  public void walk(){
    System.out.println("Berjalan dengan 2 kaki.");
  }
}

class Cat extends LivingThing
{
  /** Mengimplementasikan metode abstrak */
  public void walk(){
    System.out.println("Berjalan dengan 4 kaki.");
  }
}

public class MakhlukHidup{
  public static void main(String args[]){
    Human Manusia = new Human();
```

```
     Cat Kucing = new Cat();

      System.out.println("Manusia :");
      Manusia.breath();
      Manusia.eat();
      Manusia.walk();

      System.out.println();

      System.out.println("Kucing :");
      Kucing.breath();
      Kucing.eat();
      Kucing.walk();
   }
 }
```

5.5.2. *Interface*

```
 interface Lembaga {
   /** variabel berikut adalah static dan final */
   static final String Province = "Banten";

   /** secara default, variabel berikut adalah static dan final,
walaupun tidak dinyatakan */
   static final String District = "Tangerang Selatan";

   public void setName(String Name);
   public void setAddress(String Address);
   public void setPhone(String Phone);
   public String getName();
   public String getAddress();
   public String getPhone();
 }

 interface Tingkat{
   public void setLevel(String Level);
   public String getLevel();
 }

 class University implements Lembaga,Tingkat {
   String Name, Address, Phone, Level;

   public University(String Name, String Address, String Phone, String
Level){
      this.Name = Name;
      this.Address = Address;
      this.Phone = Phone;
      this.Level = Level;
   }

   public void setName(String Name){
      this.Name = Name;
   }

   public void setAddress(String Address){
      this.Address = Address;
   }

   public void setPhone(String Phone){
      this.Phone = Phone;
```

```
   }

   public String getName(){
      return Name;
   }

   public String getAddress(){
      return Address;
   }

   public String getPhone(){
      return Phone;
   }

   public void setLevel(String Level){
      this.Level=Level;
   }

   public String getLevel(){
      return Level;
   }

   public String toString(){
      return "Name     : "+Name+"\n"+
             "Level    : "+Level+"\n"+
             "Address  : "+Address+"\n"+
             "Phone    : "+Phone+"\n"+
             "District : "+District+"\n"+
             "Province : "+Province;
   }
}

class SMA implements Lembaga,Tingkat {
   String Name, Address, Phone, Level;

   public SMA(String Name, String Address, String Phone, String Level){
      this.Name = Name;
      this.Address = Address;
      this.Phone = Phone;
      this.Level = Level;
   }

   public void setName(String Name){
      this.Name = Name;
   }

   public void setAddress(String Address){
      this.Address = Address;
   }

   public void setPhone(String Phone){
      this.Phone = Phone;
   }

   public String getName(){
      return Name;
   }

   public String getAddress(){
      return Address;
   }
```

```
   public String getPhone(){
      return Phone;
   }

   public void setLevel(String Level){
      this.Level=Level;
   }

   public String getLevel(){
      return Level;
   }

   public String toString(){
      return "Name     : "+Name+"\n"+
             "Level    : "+Level+"\n"+
             "Address  : "+Address+"\n"+
             "Phone    : "+Phone+"\n"+
             "District : "+District+"\n"+
             "Province : "+Province;
   }
 }

 public class LembagaPendidikan {

   public static void main(String args[]){
      University Unpam;
      Unpam = new University("Universitas Pamulang","Jl. Surya Kencana
No. 1","+62 21 7412566","Universitas");

      SMA SMAN2 = new SMA("SMAN 2 Kota Tangerang Selatan","Jl. Raya
Puspiptek Muncul","+62 21","SLTA");

      System.out.println("Informasi Lembaga Pendidikan :\n" +
Unpam.toString());
      System.out.println();
      System.out.println("Informasi Lembaga Pendidikan :\n" +
SMAN2.toString());
   }
 }
```

Referensi:

1. Hariyanto, Bambang, (2007), *Esensi-esensi Bahasa Pemrograman Java*, Edisi 2, Informatika Bandung, November 2007.
2. Utomo, Eko Priyo, (2009), *Panduan Mudah Mengenal Bahasa Java*, Yrama Widya, Juni 2009.
3. Tim Pengembang JENI, JENI 1-6, Depdiknas, 2007