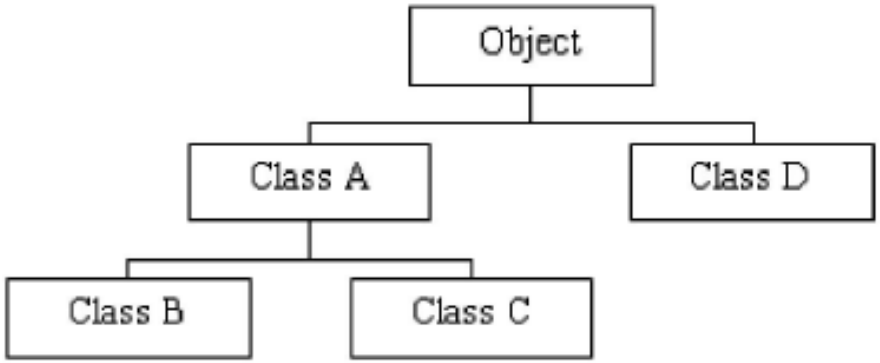
# Pewarisan, Polimorfisme, dan Interface

## Pewarisan

Dalam Java, semua class, termasuk class yang membangun Java API, adalah subclassesdari superclass Object. Contoh hirarki class diperlihatkan di bawah ini.Beberapa class di atas class utama dalam hirarki class dikenal sebagai superclass.Sementara beberapa class di bawah class pokok dalam hirarki lass dikenal sebagai subclass dari class tersebut.




Class hierarchy in Java.

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

# Pewarisan, Polimorfisme, dan Interface

## *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**. Alasannyakita melakukan ini yaitu, kita inginkan atribut-atribut ini untuk bisa diakses oleh subclassesdari 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 method dari setiap class Person yang ada. Untuk melakukan ini, kita tulis,

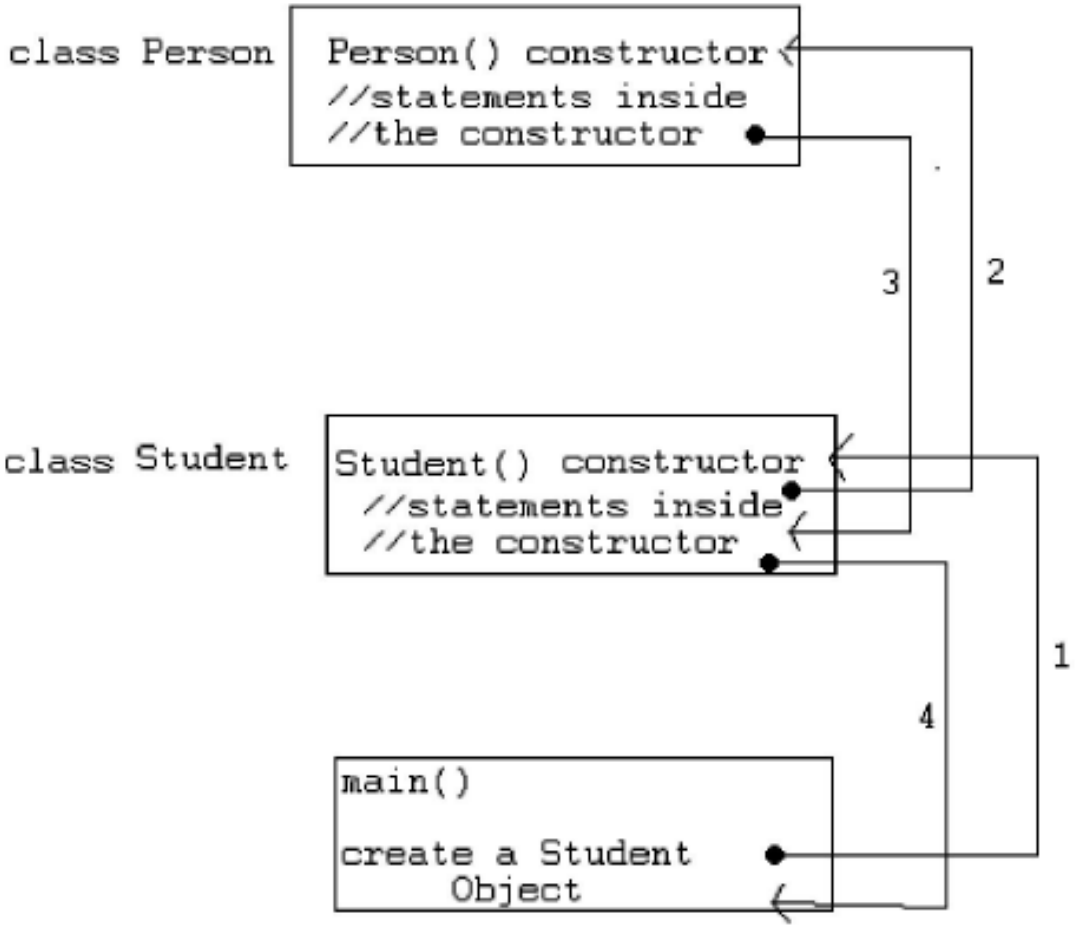
```
public class Student extends Person
{
public Student(){
System.out.println(“Inside Student:Constructor”);
//beberapa kode di sini
}
// beberapa kode di sini
}
```

Ketika object Student di-*instantiate*, default constructor dari superclass secara mutlak meminta untuk melakukan inisialisasi yang seharusnya. Setelah itu, pernyataan di dalam subclass dieksekusi. Untuk mengilustrasikannya, perhatikan kode berikut,

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

Dalam kode ini, kita membuat sebuah object dari class Student. Keluaran dari program adalah,

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



***Kata Kunci Super***

Subclass juga dapat memanggil constructor secara eksplisit dari superclass terdekat. Hal ini dilakukan dengan pemanggil construktor **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:

1.Pemanggil super() HARUS DIJADIKAN PERNYATAAN PERTAMA DALAM constructor.

2. Pemanggil super() hanya dapat digunakan dalam definisi constructor.

3. 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”;
}
```

***Overriding Method***

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

Misalnya kita mempunyai implementasi berikut untuk method getName dalam superclass Person,

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

Untuk override, method 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 method getName dari object class Student, methodoverriddeakan dipanggil, keluarannya akan menjadi,Student: getName

***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.

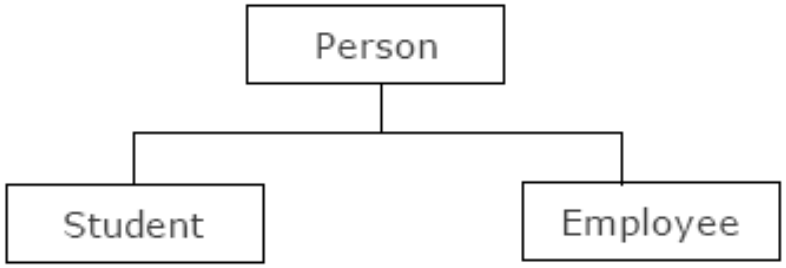
Ini memungkinkan dalam Java membuat method yang tidak dapat di-*override.* Method inidapat kita panggil **method final**. Untuk mendeklarasikan method untuk menjadi final, kita tambahkan kata kunci final ke dalam deklarasi method. Contohnya, jika kita ingin method getName dalam class Person untuk dideklarasikan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

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

Method static juga secara otomatis final. Ini artinya Anda tidak dapat membuatnyaoverride.

**Polimorfisme**

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




Dalam Java, kita dapat membuat referensi yang merupakan tipe dari superclass ke sebuah object 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 method getName dalam superclass Person kita, dan kita override method 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 method utama kita, ketika kita mencoba memanggil method getName dari reference Person ref, method getName dari object Student akan dipanggil. Sekarang, jika kita berikan ref ke object Employee, method 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 object apa yang dijadikan acuan dinamakan polimorfisme. Polimorfisme menyediakan multiobject dari subclasses yang berbeda untuk diperlakukan sebagai object dari superclass tunggal, secara otomatis menunjuk method yang tepat untuk

menggunakannya ke *particular* object berdasar subclass yang termasuk di dalamnya.

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

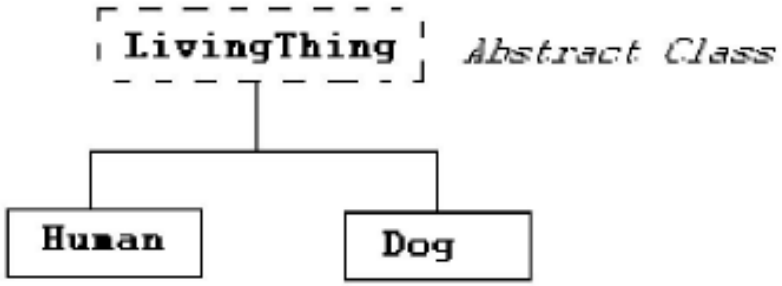
```
public static main( String[] args )
{
Student studentObject = new Student();
Employee employeeObject = new Employee();
printInformation( studentObject );
printInformation( employeeObject );
}
public static printInformation( Person p ){
. . . .
}
```

**Abstract Class**

Misalnya kita ingin membuat superclass yang mempunyai method tertentu yang berisiimplementasi, dan juga beberapa method yang akan di-overridden oleh subclasses nya. Sebagai contoh, kita akan membuat superclass bernama LivingThing. class ini mempunyai method tertentu seperti breath, eat, sleep, dan walk. Akan tetapi, ada beberapa method di dalam superclass yang sifatnya tidak dapat digeneralisasi.

Kita ambil contoh, method walk.

*Tidak semua kehidupan berjalan(walk) dalam cara yang sama.* Ambil manusia sebagai misal, kita manusia berjalan dengan dua kaki, dimana kehidupan lainnya seperti anjing berjalan dengan empat kaki. Akan tetapi, beberapa ciri umum dalam kehidupan sudah biasa, itulah kenapa kita inginkan membuat superclass umum dalam hal ini.




Kita dapat membuat superclass yang mempunyai beberapa method dengan implementasi sedangkan yang lain tidak. Class jenis ini yang disebut dengan class abstract. Sebuah **class abstract** adalah class yang tidak dapat di-instantiate. Seringkali muncul di atas hirarki class pemrograman berbasis object, dan mendefinisikan keseluruhan aksi yang mungkin pada object dari seluruh subclasses dalam class. Method ini dalam class abstract yang tidak mempunyai implementasi dinamakan method abstract. Untuk membuat method abstract, tinggal menulis deklarasi method tanpa tubuh class dan digunakan menggunakan kata kunci abstract. Contohnya,
public **abstract** void someMethod();

Sekarang mari membuat contoh class abstract.

```
public abstract class LivingThing
{
public void breath(){
System.out.println("Living Thing breathing...");
}
public void eat(){
System.out.println("Living Thing eating...");
}
/**
* abstract method walk
* Kita ingin method ini di-overridden oleh subclasses
*/
public abstract void walk();
}
```

Ketika class meng-*extend* class abstract LivingThing, dibutuhkan untuk override method abstract walk(), atau lainnya, juga subclass akan menjadi class abstract, oleh karena itu tidak dapat di-instantiate. Contohnya,

```
public class Human extends LivingThing
{
public void walk(){
System.out.println("Human walks...");
}}
```

Jika class Human tidak dapat overridde method walk, kita akan menemui pesan error berikut ini, Human.java:1: Human is not abstract and does not override

abstract method walk() in LivingThing

public class Human extends LivingThing

^

1 error

**Interface**

**Interface** adalah jenis khusus dari blok yang hanya berisi method signature(atau constant). Interface mendefinisikan sebuah(signature) dari sebuah kumpulan method 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 bahwainterface-interface juga menunjukkan polimorfisme, dikarenakan program dapat memanggil method interface dan versi yang tepat dari method yang akan dieksekusi tergantung daritipe object yang melewati pemanggil method interface.

***Kenapa Kita Memakai Interface?***

Kita akan menggunakan interface jika *kita ingin class yang tidak berhubungan mengimplementasikan method yang sama*. Melalui interface-interface, kita dapatmenangkap kemiripan diantara class yang tidak berhubungan tanpa membuatnya seolaholah class yang berhubungan. Mari kita ambil contoh class **Line** dimana berisi method yang menghitung panjang dari garis dan membandingkan object **Line** ke object dari class yang sama. Sekarang, misalkan kita punya class yang lain yaitu **MyInteger** dimana berisi method yang membandingkan object **MyInteger** ke object dari class yang sama. Seperti yang kita lihat disini, kedua class-class mempunyai method yang mirip dimana membandingkan mereka dari object lain dalam tipe yang sama, tetapi mereka tidak berhubungan sama sekali. Supaya dapat menjalankan cara untuk memastikan bahwa dua class-class ini mengimplementasikan beberapa method dengan tanda yang sama, kita dapat menggunakan sebuah interface untuk hal ini. Kita dapat membuat sebuah class interface, katakanlah interface **Relation** dimana mempunyai deklarasi method 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 object adalah *untuk menyatakan sebuah interface pemrograman object 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 object lain seperti C++.

***Interface vs. Class Abstract***

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

***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, Anda tidak dapat membuat instance dari sebuah interface.

Contohnya:

PersonInterface pi = new PersonInterface(); //COMPILE

//ERROR!!!

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

***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 object menurut urutan asli dari object.

```
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 class Anda mencoba mengimplementasikan sebuah interface, selalu pastikan bahwa Anda mengimplementasikan semua method dari interface, jika tidak, Anda akan menemukan kesalahan 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

***Hubungan dari Interface ke Class***

Seperti yang telah kita lihat dalam bagian sebelumnya, class dapat mengimplementasikansebuah interface selama kode implementasi untuk semua method yang didefinisikan dalam interface tersedia.

Hal lain yang perlu dicatat tentang hubungan antara interface ke class-class yaitu, class hanya dapat mengEXTEND 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.

***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 method dalam PersonInterface.

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