Bab 9. Inheritance dan Interface

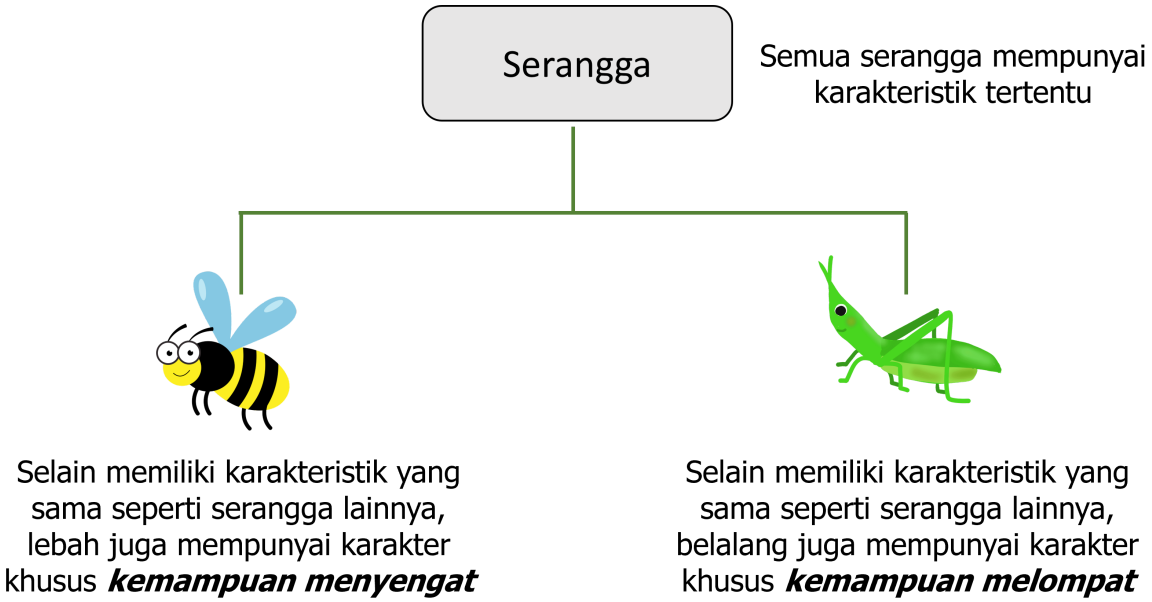
OBJEKTIF :

1. Mahasiswa mampu memahami mengenai materi Inheritance dan Interface pada Java.
2. Mahasiswa mampu memahami mengenai penggunaan Inheritance dan Interface pada program Java.
3. Mahasiswa mampu mensimulasikan penggunaan Inheritance dan Interface pada program Java untuk kejadian di dunia nyata.

9.1 Inheritance

Generalisasi dan Spesialisasi

Dalam dunia nyata, kita dapat melihat banyak objek-objek yang merupakan versi spesialisasi dari objek yang lebih umum. Sebagai contoh, serangga adalah jenis binatang yang mempunyai ciri-ciri tertentu. Belalang dan lebah adalah binatang-binatang yang termasuk serangga karena mereka mempunyai karakteristik-karakteristik dari serangga. Selain itu, mereka juga mempunyai karakteristik-karakteristik masing-masing. Misalkan, belalang mempunyai kemampuan untuk melompat dan lebah mempunyai kemampuan menyengat. Belalang dan lebah adalah versi spesialisasi dari serangga. Ini diilustrasikan pada gambar berikut:



Inheritance dan Relasi "Is a"

Object yang merupakan versi spesialisasi dari suatu object mempunyai relasi "Is a" (adalah sebuah) dengan object yang merupakan versi generalisasinya. Misalkan, belalang "is a" (adalah seekor) serangga. Contoh-contoh lain relasi "is a":

- Mobil "is a" (adalah sebuah) kendaraan.
- Bunga "is a" (adalah sebuah) tumbuhan.
- Persegi panjang "is a" (adalah sebuah) bentuk.
- Pemain bulu tangkis "is a" (adalah seorang) atlet.

Jika terdapat sebuah relasi "is-a" diantara dua object, ini berarti object versi spesialisasi mempunyai semua karakteristik dari object versi umumnya, dan juga karakteristik tambahan yang membuatnya lebih khusus. Dalam pemrograman berorientasi object, inheritance (pewarisan) digunakan untuk membuat sebuah relasi "is -a" antara class-class. Inheritance memungkinkan kita untuk mengekstensi kemampuan dari sebuah class dengan membuat class lain yang merupakan versi spesialisasi dari class tersebut.

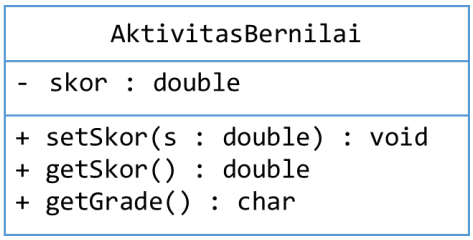
Superclass dan Subclass

Pada inheritance, class yang merupakan versi spesialisasi dari suatu class disebut dengan **subclass** (atau class turunan). Sedangkan class yang merupakan versi umum dari suatu class disebut dengan **superclass** (atau class dasar). Subclass mewarisi field-field dan method-method dari superclass. Selain itu, subclass juga mempunyai field-field dan method-method lain yang membuatnya versi spesialisasi dari superclass.

Contoh Inheritance

Kita akan melihat bagaimana inheritance digunakan dalam Java. Kita akan membuat sebuah class yang menyimpan nilai tugas-tugas mahasiswa.

Kita akan membuat sebuah program yang mengelola tugas-tugas yang dinilai dengan skor angka seperti 70, 85, 90, dsb., dan grade huruf seperti A, B, C, D, atau E. Misalkan tugas yang dinilai ini direpresentasikan dengan sebuah class bernama `AktivitasBernilai`. Class ini menyimpan skor angka dan grade huruf dari aktivitas yang dinilai. Misalkan UML dari class `AktivitasBernilai` adalah seperti gambar berikut:



Pada diagram UML di atas, dapat dilihat class `AktivitasBernilai` mempunyai sebuah field tipe `double` bernama `skor`. Class ini mempunyai tiga method: method `setSkor` yang digunakan untuk menetapkan skor angka, method `getSkor` yang mengembalikan skor angka, dan method `getGrade` yang mengembalikan grade huruf dari skor angka. Perhatikan juga, class `AktivitasBernilai` tidak mempunyai constructor, sehingga Java akan membuatkan constructor default untuk class ini. Kode berikut adalah kode untuk definisi class `AktivitasBernilai`:

Definisi Class `AktivitasBernilai` (AktivitasBernilai.java)

```
/*
    Class yang menyimpan nilai untuk aktivitas bernilai.
*/
public class AktivitasBernilai
{
    private double skor;    // Skor numerik

    /*
        Method setSkor menetapkan nilai field skor.
        @param s Nilai untuk disimpan dalam skor.
    */
    public void setSkor(double s)
    {
        skor = s;
    }

    /*
        Method getSkor mengembalikan nilai field skor.
        @return Nilai yang disimpan dalam field skor.
    */
    public double getSkor()
    {
        return skor;
    }

    /*
        Method getGrade mengembalikan grade huruf
        yang ditentukan dari nilai pada field skor.
        @return Grade huruf.
    */
    public char getGrade()
    {
        char gradeHuruf;

        if (skor >= 90)
        {
            gradeHuruf = 'A';
        }
        else if (skor >= 80)
        {
            gradeHuruf = 'B';
        }
        else if (skor >= 70)
        {
            gradeHuruf = 'C';
        }
        else if (skor >= 60)
        {
            gradeHuruf = 'D';
        }
        else
        {
            gradeHuruf = 'E';
        }
```

```
        return gradeHuruf;
    }
}
```

Program berikut mendemonstrasikan penggunaan class `AktivitasBernilai`:

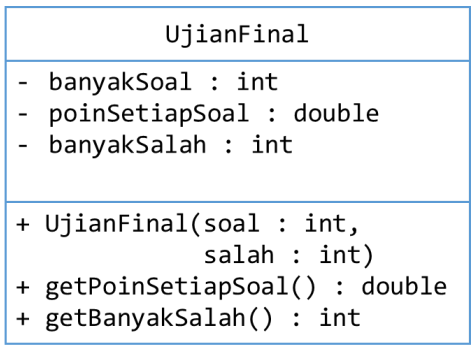
Program (DemoGrade.java)

```
import java.util.Scanner;

/*
    Program ini mendemonstrasikan
    class AktivitasBernilai.
*/
public class DemoGrade
{
    public static void main(String[] args)
    {
        double skor;

        // Buat object AktivitasBernilai
        AktivitasBernilai ujian = new AktivitasBernilai();

        // Minta skor ke pengguna
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Masukkan skor ujian: ");
        skor = keyboard.nextDouble();

        // Simpan skor dalam object ujian.
        ujian.setSkor(skor);

        // Tampilkan grade huruf dari skor
        System.out.println("Grade untuk ujian adalah " + ujian.getGrade());
    }
}
```

Output Program (DemoGrade.java)

```
Masukkan skor ujian: 87
Grade untuk ujian adalah B
```

Class `AktivitasBernilai` merepresentasikan karakterisik umum dari aktivitas yang dinilai dari seorang siswa. Aktivitas bernilai ini dapat berupa berbagai jenis aktivitas pada sebuah mata kuliah seperti kuis, ujian midterm, ujian final, praktikum, tugas essay, dan sebagainya. Berbagai jenis dari aktivitas yang dinilai ini masing-masing dapat direpresentasikan sebagai subclass dari class `AkvitasBernilai`. Sebagai contoh, kita dapat membuat subclass dari `AktivitasBernilai` yang bernama `UjianFinal` untuk merepresentasikan ujian final. Gambar berikut adalah diagram UML untuk class `UjianFinal`:



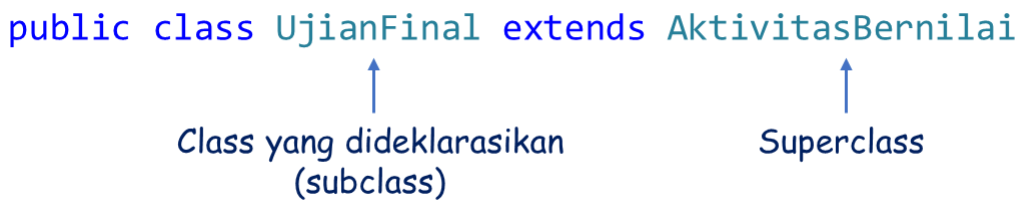
Pada diagram UML di atas, class `UjianFinal` mempunyai field untuk banyaknya soal pada ujian (`bykSoal`), nilai poin setiap soal (`poinSoal`), dan banyaknya soal yang dijawab salah oleh siswa (`bykSalah`).

Kode berikut adalah definisi dari subclass `UjianFinal`:

```
/*
    Class ini menentaukan grade untuk ujian final.
    Class ini adalah subclass dari class AktivitasBernilai.
*/
public class UjianFinal extends AktivitasBernilai
{
    private int banyakSoal;         // Banyak soal dalam ujian final
    private double poinSetiapSoal;  // Poin untuk setiap soal
    private int banyakSalah;        // Banyak soal yang dijawab salah

    /*
        Constructor ini menetapkan banyak soal pada ujian dan banyak
        soal yang dijawab salah.
        @param soal  Banyak soal dalam ujian.
        @param salah Banyak soal yang dijawab salah.
    */
    public UjianFinal(int soal, int salah)
    {
        double skorNumerik;         // Untuk menyimpan skor numerik

        // Tetapkan nilai pada field banyakSoal dan field banyakSalah.
        banyakSoal = soal;
        banyakSalah = salah;

        // Hitung poin untuk setiap soal dan skor numerik untuk ujian ini.
        poinSetiapSoal = 100.0 / soal;
        skorNumerik = 100.0 - (salah * poinSetiapSoal);

        // Panggil method setSkor yang diwarisi untuk
        // menetapkan skor numerik.
        setSkor(skorNumerik);
    }

    /*
        Method getPoinSetiapSoal mengembalikan banyaknya poin dari
        setiap soal.
        @return Nilai dalam field poinSetiapSoal.
    */
    public double getPoinSetiapSoal()
```

```
    {
        return poinSetiapSoal;
    }

    /*
        Method getBanyakSalah mengembalikan banyaknya soal
        yang dijawab salah.
        @return Nilai dalam field banyakSalah.
    */
    public int getBanyakSalah()
    {
        return banyakSalah;
    }
}
```

Perhatikan header dari class `UjianFinal` pada baris 5. Header ini menggunakan keyword `extends` yang menandakan bahwa class ini mengekstensi class lain (sebuah superclass). Nama dari superclass dituliskan setelah keyword `extends`. Sehingga, baris 5 ini, menandakan bahwa `UjianFinal` adalah nama dari class yang dideklarasikan dan `AktivitasBernilai` adalah nama superclass yang diekstensi oleh class ini.



Karena class `UjianFinal` adalah subclass dari class `AktivitasBernilai` maka class `UjianFinal` mewarisi semua member-member public dari class `AktivitasBernilai`. Sehingga class `UjianFinal` akan mempunyai member-member berikut:

Field-field:

`int banyakSoal;`	Dideklarasikan di `UjianFinal`
`double poinSetiapSoal;`	Dideklarasikan di `UjianFinal`
`int banyakSalah;`	Dideklarasikan di `UjianFinal`

Method-method:

Constructor	Dideklarasikan di `UjianFinal`
`getPoinSetialSoal`	Dideklarasikan di `UjianFinal`
`getBanyakSalah`	Dideklarasikan di `UjianFinal`
`setSkor`	Diwariskan dari `AktivitasBernilai`
`getSkor`	Diwariskan dari `AktivitasBernilai`
`getGrade`	Diwariskan dari `AktivitasBernilai`

Perhatikan bahwa field `skor` dari class `AktivitasBernilai` tidak ada di daftar member-member class `UjianFinal` di atas. Ini karena field `skor` adalah private. Member-member private dari superclass tidak dapat diakses oleh subclass, sehingga mereka tidak diwarisi. Ketika sebuah object dari subclass dibuat, member-member private dari superclass juga dibuat di memori, tetapi hanya method-method yang berada di dalam superclass yang dapat mengaksesnya.

Perhatikan juga bahwa constructor superclass tidak terdapat di member-member dari class `UjianFinal`. Ini karena constructor superclass hanya mempunyai tujuan untuk mengkonstruksi object-object dari superclass sehingga constructor superclass tidak diwarisi.

Untuk melihat bagaimana inheritance bekerja dalam contoh ini, perhatikan constructor dari `UjianFinal` pada baris 17 sampai dengan 32:

```
public UjianFinal(int soal, int salah)
{
    double skorNumerik;          // Untuk menyimpan skor numerik

    // Tetapkan nilai pada field banyakSoal dan field banyakSalah.
    banyakSoal = soal;
    banyakSalah = salah;

    // Hitung poin untuk setiap soal dan skor numerik untuk ujian ini.
    poinSetiapSoal = 100.0 / soal;
    skorNumerik = 100.0 - (salah * poinSetiapSoal);

    // Panggil method setSkor yang diwarisi untuk
    // menetapkan skor numerik.
    setSkor(skorNumerik);
}
```

Constructor ini menerima dua argument: banyaknya soal dalam ujian dan banyaknya soal yang dijawab salah oleh siswa. Pada baris 22 dan 23, nilai-nilai ini ditugaskan ke field `banyakSoal` dan field `banyakSalah`. Lalu, pada baris 26 dan 27, banyaknya poin untuk setiap soal dan skor numerik ujian dihitung. Pada baris 31, statement terakhir dalam constructor adalah:

```
setSkor(skorNumerik);
```

Statement di atas adalah pemanggilan method `setSkor`. Meskipun tidak ada method `setSkor` dalam class `UjianFinal`, method ini diwarisi dari class `AktivitasBernilai`. Sehingga method ini dapat dipanggil di dalam class `UjianFinal`. Program berikut mendemonstrasikan class `UjianFinal`:

Program (DemoUjianFinal)

```
import java.util.Scanner;

/*
    Program ini mendemonstrasikan class UjianFinal
    yang mengekstensi class AktivitasBernilai.
*/
public class DemoUjianFinal
{
    public static void main(String[] args)
    {
        int bykSoal;        // Banyak soal dalam ujian
        int bykSalah;      // Banyak soal dijawab salah

        Scanner keyboard = new Scanner(System.in);
        System.out.print("Berapa banyak soal dalam ujian final? ");
        bykSoal = keyboard.nextInt();

        System.out.print("Berapa banyak soal yang dijawab salah oleh siswa? ");
        bykSalah = keyboard.nextInt();

        // Buat object UjianFinal
        UjianFinal ujian = new UjianFinal(bykSoal, bykSalah);

        // Tampilkan hasil ujian
        System.out.println("Poin setiap soal = " + ujian.getPoinSetiapSoal());
        System.out.println("Skor ujian siswa ini = " + ujian.getSkor());
        System.out.println("Grade ujian siswa ini = " + ujian.getGrade());
    }
}
```

Output Program (DemoUjianFinal.java)

```
Berapa banyak soal dalam ujian final? 20
Berapa banyak soal yang dijawab salah oleh siswa? 3
Poin setiap soal = 5.0
Skor ujian siswa ini = 85.0
Grade ujian siswa ini = B
```

Perhatikan pada baris 22, statement berikut membuat instance dari class `UjianFinal` dan menugaskan alamatnya ke variabel `ujian`:

```
UjianFinal ujian = new UjianFinal(bykSoal, bykSalah);
```

Ketika object `UjianFinal` dibuat dalam memori, object tersebut tidak hanya mempunyai member-member yang dideklarasikan dalam class `UjianFinal`, tetapi juga member-member non-private yang dideklarasikan dalam class `AktivitasBernilai`. Perhatikan pada baris 26 dan 27, dua method public dari class `AktivitasBernilai`, method `getSkor` dan method `getGrade` dipanggil langsung dari object `ujian`:

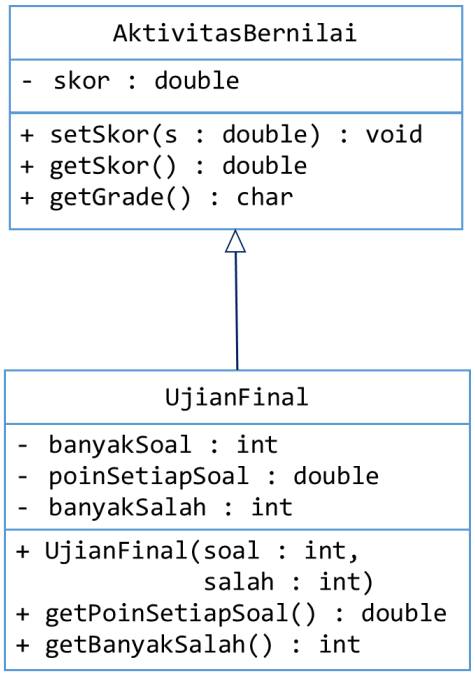
```
System.out.println("Skor ujian siswa ini = " + ujian.getSkor());
System.out.println("Grade ujian siswa ini = " + ujian.getGrade());
```

Ketika subclass mengektensi superclass, member-member public dari superclass menjadi member-member public dari subclass. Dalam program di atas, method `getSkor` dan method `getGrade` dapat dipanggil dari object `ujian` karena method-method tersebut adalah member-member public dari superclass dari object.

Seperti yang telah disebutkan sebelumnya, member-member private dari superclass (dalam contoh ini field `skor`) tidak dapat diakses oleh subclass. Ketika object `ujian` dibuat dalam memori, field `skor` juga dibuat dalam memori. Namun, hanya method-method yang didefinisikan dalam superclass `AktivitasBenilai` yang dapat mengakses field `skor` ini. Karena constructor `UjianFinal` tidak dapat mengakses field `skor`, maka constructor ini harus memanggil method `setSkor` dari superclass untuk menyimpan nilai ke field tersebut.

Inheritance dalam Diagram UML

Kita menunjukkan relasi inheritance pada diagram UML dengan menghubungkan kedua class dengan sebuah garis dengan ujung panah. Ujung panah tersebut digambarkan mengarah ke superclass. Gambar berikut adalah diagram UML yang menunjukkan relasi antara class `AktivitasBernilai` dan `UjianFinal`:



Constructor Superclass

Dalam relasi inheritance, constructor superclass selalu dieksekusi sebelum constructor subclass. Pada contoh inheritance di atas, class `AktivitasBernilai` hanya mempunyai satu constructor yaitu constructor default yang Java tuliskan secara otomatis. Saat sebuah object `UjianFinal` dibuat, constructor default dari class `AktivitasBernilai` dieksekusi sebelum constructor `UjianFinal` dieksekusi.

Untuk memperlihatkan constructor superclass selalu dieksekusi sebelum constructor subclass dieksekusi saat pembuatan object subclass, perhatikan dua kode class berikut:

Definisi Class `SuperClass1` (SuperClass1.java)

```
public class SuperClass1
{
    /*
        Constructor
    */
    public SuperClass1()
    {
        System.out.println("Ini adalah constructor superclass.");
    }
}
```

Definisi Class `SubClass1` (SubClass1.java)

```
public class SubClass1 extends SuperClass1
{
    /*
        Constructor
    */
    public SubClass1()
    {
        System.out.println("Ini adalah constructor subclass.");
    }
}
```

Kode pertama adalah kode class `SuperClass1` yang mempunyai constructor tanpa argument. Constructor ini hanya menampilkan pesan "Ini adalah constructor superclass". Kode kedua adalah kode dari `SubClass1` yang mengekstensi `SuperClass1`. Class ini juga mempunyai constructor tanpa argument yang menampilkan pesan "Ini adalah constructor subclass".

Program berikut mendemokan pembuatan object `SubClass1`:

Program (DemoConstructor.java)

```
/*
    Program ini mendemonstrasikan urutan eksekusi
    constructor superclass dan subclass saat object
    subclass dibuat.
*/
public class DemoConstructor1
{
    public static void main(String[] args)
    {
        SubClass1 obj = new SubClass1();
    }
}
```

Output Program (DemoConstructor.java)

```
Ini adalah constructor superclass.
Ini adalah constructor subclass.
```

Seperti yang dapat Anda lihat pada output program, constructor superclass dieksekusi terlebih dahulu lalu diikuti oleh constructor subclass.

Hal yang perlu diingat mengenai constructor superclass dan constructor subclass dalam relasi inheritance adalah: jika sebuah superclass memiliki constructor default atau constructor tanpa argument yang ditulis dalam class, maka constructor tersebut akan secara otomatis dipanggil sebelum constructor subclass dieksekusi. Kita akan melihat situasi-situasi lain yang melibatkan constructor superclass pada bagian berikutnya.

9.2 Memanggil Constructor Superclass

Pada bagian sebelumnya, kita telah melihat contoh-contoh yang menunjukkan bagaimana constructor default atau constructor tanpa argument dari superclass secara otomatis dipanggil sebelum constructor subclass dieksekusi. Bagaimana jika superclass tidak mempunyai default constructor atau constructor tanpa argument? Atau jika superclass mempunyai lebih dari satu constructor ter-overloading dan kita ingin memastikan satu constructor tertentu yang dipanggil? Dalam situasi-situasi seperti ini, kita menggunakan keyword `super` untuk memanggil constructor superclass secara eksplisit. Keyword `super` merujuk ke superclass dari object dan dapat digunakan untuk mengakses member-member dari superclass.

Untuk memperlihatkan penggunaan keyword `super` perhatikan dua kode class berikut:

Definisi Class `SuperClass2` (SuperClass2.java)

```
public class SuperClass2
{
    /*
        Constructor #1
    */
    public SuperClass2()
    {
        System.out.println("Ini adalah constructor tanpa argument superclass");
    }

    /*
        Constructor #2
    */
    public SuperClass2(int arg)
    {
        System.out.println("Argument berikut diberikan " +
                           "ke constructor superclass: " + arg);
    }
}
```

Definisi Class `SubClass2` (SubClass2.java)

```
public class SubClass2 extends SuperClass2
{
    /*
        Constructor
    */
    public SubClass2()
    {
        super(10);
        System.out.println("Ini adalah constructor subclass.");
    }
}
```

Perhatikan kode dari `SubClass2`. Pada baris 8 di dalam constructor `SubClass2` kita menuliskan statement berikut:

```
super(10);
```

Statement ini memanggil constructor superclass dengan memberikan nilai 10 sebagai argument. Terdapat tiga ketentuan dalam pemanggilan constructor superclass:

- Statement `super` yang memanggil constructor superclass hanya dapat dituliskan di dalam constructor subclass. Kita tidak dapat memanggil constructor superclass dari method-method lainnya.
- Statement `super` yang memanggil constructor superclass haruslah statement pertama dalam constructor subclass. Ini karena constructor superclass harus dieksekusi sebelum kode dalam constructor subclass dieksekusi.
- Jika constructor subclass tidak secara eksplisit memanggil constructor superclass, Java akan secara otomatis memanggil constructor default dari superclass, atau memanggil constructor tanpa argument dari superclass sebelum mengeksekusi kode-kode di dalam constructor subclass.

Program berikut mendemonstrasikan kedua class di atas:

Program (DemoConstructor2.java)

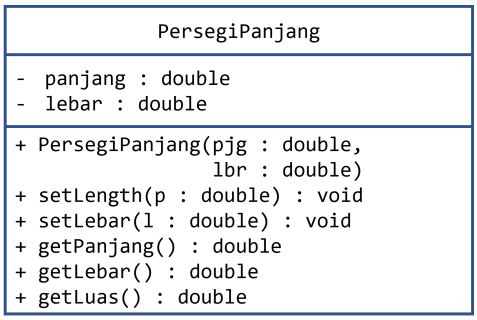
```
/*
    Porgram ini mendemonstrasikan bagaimana constructor
    superclass dipanggil dengan keyword super.
*/
public class DemoConstructor2
{
    public static void main(String[] args)
    {
        SubClass2 obj = new SubClass2();
    }
}
```

Output Program (DemoConstructor2.java)

```
Argument berikut diberikan ke constructor superclass: 10
Ini adalah constructor subclass.
```

Contoh Lain Pemanggilan Constructor Superclass

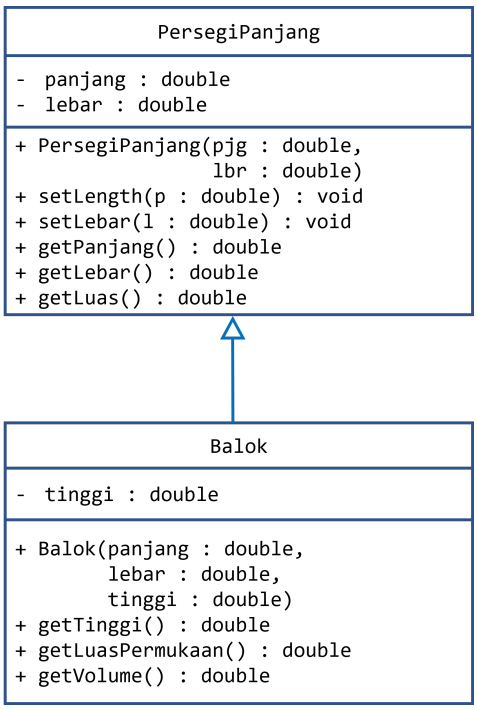
Kita akan menggunakan class `PersegiPanjang` yang kita buat pada topik sebelumnya. Gambar berikut adalah diagram UML dari class `PersegiPanjang`:



Berikut adalah potongan dari kode class `PersegiPanjang`:

```
/*
     Class PersegiPanjang dengan constructor.
*/
public class PersegiPanjang
{
    private double panjang;
    private double lebar;

    /*
        Constructor
        @param pjg Panjang dari persegi panjang.
        @param lbr Lebar dari persegi panjang.
    */
    public PersegiPanjang(double pjg, double lbr)
    {
        panjang = pjg;
        lebar = lbr;
    }

    ...terdapat kode-kode lain yang tidak ditampilkan disini.
}
```

Selanjutnya kita akan membuat class `Balok`, yang mengekstensi class `PersegiPanjang`. Class `Balok` didesain untuk menyimpan data mengenai balok, yang tidak hanya mempunyai panjang, lebar, dan luas (luas dasar), tapi juga mempunyai tinggi, luas permukaan, dan volume. Gambar berikut adalah diagram UML yang menunjukkan relasi inheritance dari class `PersegiPanjang` dan class `Balok`:



Kode berikut adalah definisi class `Balok`:

Definisi Class `Balok` (Balok.java)

```
public class Balok extends PersegiPanjang
{
    private double tinggi;      // Tinggi balok

    /*
        Constructor menetapkan panjang, lebar, dan
        tinggi dari balok.
        @param panjang Panjang balok.
        @param lebar Lebar balok.
        @param tinggi Tinggi balok.
    */
    public Balok(double panjang, double lebar, double tinggi)
    {
        // Panggil constructor superclass
        super(panjang, lebar);

        // Tetapkan filed tinggi.
        this.tinggi = tinggi;
    }

    /*
        Method getTinggi mengembalikan tinggi balok.
        @return Nilai dalam field tinggi.
    */
    public double getTinggi()
    {
        return tinggi;
    }

    /*
        Method getLuasPermukaan menghitung dan
        mengembalikan luas permukaan balok.
```

```
        @return Luas permukaan balok.
    */
    public double getLuasPermukaan()
    {
        return 2 * getLuas() +
               2 * getPanjang() * tinggi +
               2 * getLebar() * tinggi;
    }

    /*
        Method getVolume menghitung dan
        mengembalikan volume balok.
     */
    public double getVolume()
    {
        return getLuas() * tinggi;
    }
}
```

Pada kode di atas, kita mendefinisikan constructor `Balok` untuk menerima argument-argument untuk parameter-parameter `panjang`, `lebar`, dan `tinggi`. Nilai-nlai yang diberikan ke `panjang` dan `lebar` kita berikan lagi sebagai argument ke constructor `PersegiPanjang` pada baris 15:

```
super(panjang, lebar);
```

Setelah constructor `PersegiPanjang` selesai dieksekusi, kode-kode berikutnya dalam constructor `Balok` kemudian dieksekusi.

Program berikut mendemonstrasikan class `Balok` ini:

Program (DemoBalok.java)

```
import java.util.Scanner;

public class DemoBalok
{
    public static void main(String[] args)
    {
        double panjang;     // Panjang balok
        double lebar;       // Lebar balok
        double tinggi;      // Tinggi balok

        // Buat object Scanner untuk input keyboard
        Scanner keyboard = new Scanner(System.in);

        // Dapatkan dimensi balok.
        System.out.println("Masukkan dimensi dari balok!");
        System.out.print("Panjang: ");
        panjang = keyboard.nextDouble();
        System.out.print("Lebar: ");
        lebar = keyboard.nextDouble();
        System.out.print("Tinggi: ");
        tinggi = keyboard.nextDouble();

        // Buat object Balok dan berikan dimensi ke constructor.
        Balok myBalok = new Balok(panjang, lebar, tinggi);
```

```
        // Tampilkan properti-properti balok.
        System.out.println("Berikut adalah properti-properti dari balok.");
        System.out.println("Panjang = " + myBalok.getPanjang());
        System.out.println("Lebar = " + myBalok.getLebar());
        System.out.println("Tinggi = " + myBalok.getTinggi());
        System.out.println("Luas Permukaan = " + myBalok.getLuasPermukaan());
        System.out.println("Volume = " + myBalok.getVolume());
    }
}
```

Output Program (DemoBalok.java)

```
Masukkan dimensi dari balok!
Panjang: 5
Lebar: 2
Tinggi: 3
Berikut adalah properti-properti dari balok.
Panjang = 5.0
Lebar = 2.0
Tinggi = 3.0
Luas Permukaan = 62.0
Volume = 30.0
```

Hal yang perlu dicatat pada contoh di atas adalah class `PersegiPanjang` mempunyai satu constructor yang menerima dua argument. Karena terdapat constructor pada class `PersegiPanjang` maka Java tidak menyediakan secara otomatis constructor default. Sehingga, pada class `PersegiPanjang` tidak terdapat constructor tanpa argument ataupun constructor default. Jika superclass tidak mempunyai constructor default atau constructor tanpa argument, maka class yang mengekstensinya harus memanggil salah satu constructor yang dipunyai oleh superclass. Jika tidak, error kompilasi akan didapatkan.

Ringkasan Hal-Hal Penting Mengenai Constructor pada Inheritance

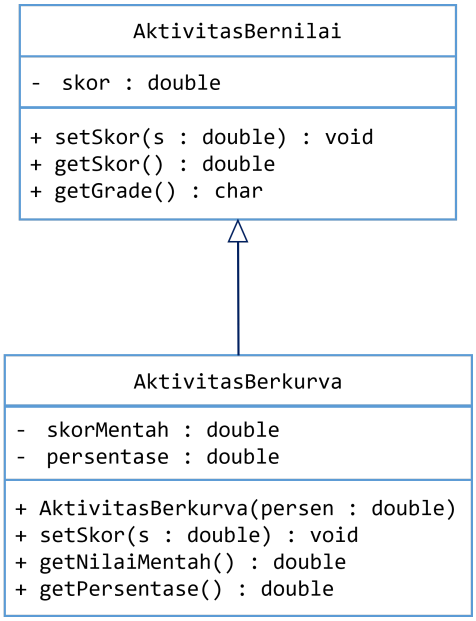
Berikut adalah ringkasan hal-hal penting yang perlu kita ketahui mengenai constructor pada inheritance:

- Constructor superclass selalu dieksekusi sebelum constructor subclass.
- Kita dapat menuliskan statement `super` yang memanggil constructor superclass, tetapi hanya pada constructor subclass. Kita tidak dapat memanggil construcotr superclass pada method-method lain.
- Jika statement `super` memanggil constructor superclass terdapat di dalam constructor sublcass, statement tersebut harus sebagai statement pertama.
- Jika constructor subclass tidak secara eksplisit memanggil constructor superclass, Java akan secara otomatis memanggil `super()` sebelum mengeksekusi kode-kode di dalam constructor subclass.
- Jika superclass tidak mempunyai constructor default maupun constructor tanpa argument, maka class yang mengekstensinya harus memanggil salah satu constructor yang dimiliki oleh superclass.

9.3 Method Overriding

Subclass mewarisi method-method dari superclass. Jika dari method-method yang diwarisi terdapat beberapa method-method yang tidak cukup atau tidak cocok untuk tujuan dari subclass, kita dapat menggantikan method-method yang tidak cocok ini dengan meng-overriding (menimpa) method-method tersebut. Untuk meng-overriding suatu method superclass, kita menuliskan method dengan signature yang sama dengan method superclass yang ingin di-overriding.

Sebagai contoh, class `AktivitasBernilai` yang kita tulis pada bagian awal, mempunyai method `setSkor` yang menetapkan skor numerik dan method `getGrade` yang mengembalikan grade huruf berdasarkan skor numerik tersebut. Misalkan, seorang guru ingin memberikan nilai dengan kurva untuk suatu ujian sebelum nilai huruf ditentukan. Nilai dengan kurva ini dihitung dengan mengalikan setiap skor ujian siswa dengan suatu persentase. Lalu nilai yang telah dikalikan dengan suatu persentase ini digunakan untuk menentukan grade huruf. Untuk mengakomodasi ini, kita mendesain sebuah class baru, `AktivitasBerkurva` yang mengekstensi class `AktivitasBernilai` dan mempunyai versi spesialisasi sendiri dari method `setSkor`. Method `setSkor` dalam subclass meng-overriding method `setSkor` dalam superclass. Gambar berikut adalah diagram UML yang menunjukkan relasi antara class `AktivitasBernilai` dan class `AktivitasBerkurva`:



Tabel berikut menjelaskan field-field dari class `AktivitasBerkurva`:

Field	Keterangan
`skorMentah`	Field ini menyimpan skor siswa sebenarnya.
`persentase`	Field ini menyimpan persentase untuk dikalikan dengan skor siswa sebenarnya untuk mendapatkan skor terkurva.

Tabel berikut menjelaskan method-method dari class `AktivitasBerkurva`:

Method	Keterangan
Constructor	Constructor ini menerima sebuah argument bertipe `double` yang merupakan persentase kurva. Nilai argument ini ditugaskan ke field `persentase` dan field `skorMentah` ditugaskan dengan nilai 0.0.
`setSkor`	Method ini meng-overriding method `setSkor` dalam superclass. Method ini menerima sebuah argument bertipe `double` yang merupakan skor siswa sebenarnya. Method ini menyimpan argument tersebut ke field `nilaiMentah`, dan lalu memberikan hasil dari `nilaiMentah * persentase` sebagai argument ke method `setSkor` dari superclass.
`getNilaiMentah`	Method ini mengembalikan nilai dalam field `nilaiMentah`.
`getPersentase`	Method ini mengembalikan nilai dalam field `persentase`.

Kode berikut adalah kode dari class `AktivitasBerkurva`:

Definisi Class `AktivitasBerkurva` (AktivitasBerkurva.java)

```
/*
    Class ini menghitung grade berkurva.
    Class ini mengekstensi class AktivitasBernilai.
*/
public class AktivitasBerkurva extends AktivitasBernilai
{
    private double nilaiMentah;     // Skor sebenarnya
    private double persentase;      // Persentase kurva

    /*
        Constructor menetapkan persentase kurva pada field persentase
        dan menetapkan nilai 0.0 ke field nilaiMentah.
        @param persen Persentase kurva.
    */
    public AktivitasBerkurva(double persen)
    {
        persentase = persen;
        nilaiMentah = 0.0;
    }

    /*
        Method setSkor meng-overriding method setSkor superclass.
        Versi ini menerima skor mentah sebagai argument. Skor tersebut
        lalu dikalikan dengan persentase kurva dan hasilnya diberikan sebagai
        argument ke method setSkor dari superclass.
        @param s Skor mentah.
    */
    @Override
    public void setSkor(double s)
    {
        nilaiMentah = s;
        super.setSkor(nilaiMentah * persentase);
    }
```

```
    /*
        Method getNilaiMentah mengembalikan nilai mentah.
        @return Nilai dalam field nilaiMentah.
    */
    public double getNilaiMentah()
    {
        return nilaiMentah;
    }

    /*
        Method getPersentase mengembalikan persentase kurva.
        @return Nilai dalam field persentase.
    */
    public double getPersentase()
    {
        return persentase;
    }
}
```

Perhatikan pada baris 28 sampai dengan 33, kita mempunyai kode seperti berikut:

```
@Override
public void setSkor(double s)
{
    nilaiMentah = s;
    super.setSkor(nilaiMentah * persentase);
}
```

Sebelum kita membahas mengenai `@Override` pada baris 28, kita akan melihat definisi method `setSkor` pada baris 29 sampai dengan 33 terlebih dahulu. Pada baris 29, header dari method `setSkor` kita tuliskan seperti berikut:

```
public void setSkor(double s)
```

Header method `setSkor` dalam class `AktivitasBerkurva` mempunyai signature:

```
setSkor(double)
```

Signature method ini sama dengan signature method `setSkor` dalam class `AkvitasBernilai`. Sehingga, method `setSkor` dalam class `AktivitasBerkurva` ini meng-overriding method `setSkor` dalam class `AktivitasBernilai`. Method ini menerima sebuah argument bertipe `double` yang merupakan skor mentah. Kemudian, pada baris 31, statement:

```
nilaiMentah = s;
```

menugaskan nilai yang diterima oleh parameter `s` ke field `nilaiMentah`. Lalu, pada baris 32, terdapat statement berikut:

```
super.setSkor(nilaiMentah * persentase);
```

Keyword `super` pada statement ini merujuk ke object superclass. Sehingga, statement ini memanggil method `setSkor` versi superclass dengan memberikan argument berupa hasil dari ekspresi `nilaiMentah * persentase`. Kita memanggil method `setSkor` dari superclass karena kita ingin menyimpan skor ini pada field `skor` dari superclass. Dan karena field `skor` adalah private maka subclass tidak bisa mengakses langsung field ini, sehingga kita harus menggunakan method `setSkor` dari superclass.

Sebelum definisi method `setSkor` yang meng-overriding method `setSkor` superclassnya, pada baris 29, kita menuliskan `@Override`. `@Override` adalah anotasi override. Anotasi ini memberitahukan compiler Java bahwa method yang dituliskan setelahnya yaitu method `setSkor`, dimaksudkan untuk meng-overriding sebuah method dalam superclass.

Anotasi `@Override` tidak harus dituliskan, tetapi dianjurkan untuk ditulis. Dengan menuliskan anotasi ini, jika method yang ditulis setelahnya gagal meng-overriding method dalam superclass, compiler akan menampilkan error. Sebagai contoh, misalkan kita salah menuliskan header dari method pada baris 29 seperti berikut:

```
public void setskor(double s)
```

Jika kita perhatikan seksama nama dari method pada header di atas, kita akan melihat bahwa nama method ini dituliskan dalam huruf kecil semua. Nama ini tidak cocok dengan nama method dalam superclass yang ingin kita overriding, yaitu `setSkor`. Tanpa menuliskan anotasi `@Override`, kode class dengan header method seperti di atas akan berhasil dikompilasi dan dieksekusi, tetapi kita tidak akan mendapatkan hasil yang sesuai dengan yang kita inginkan karena method dalam subclass tersebut tidak meng-overriding method dalam superclass. Namun, dengan menuliskan anotasi `@Override`, compiler akan memberikan pesan error yang memberitahukan kita bahwa method subclass tidak meng-overriding method apapun dalam method superclass.

Program berikut mendemonstrasikan class `AktivitasBerkurva`:

Program (DemoAktivitasBerkurva.java)

```
import java.util.Scanner;

/*
    Program ini mendemonstrasikan class AktivitasBerkurva,
    yang merupakan subclass dari class AktivitasBernilai.
*/
public class DemoAktivitasBerkurva
{
    public static void main(String[] args)
    {
        double skor;            // Skor mentah
        double persenKurva;     // Persentase Kurva

        // Buat object Scanner untuk membaca input keyboard
        Scanner keyboard = new Scanner(System.in);

        // Dapatkan skor mentah ujian.
        System.out.print("Masukkan skor mentah siswa: ");
        skor = keyboard.nextDouble();

        // Dapatkan persentase kurva
        System.out.print("Masukkan persentase kurva: ");
```

```
        persenKurva = keyboard.nextDouble();

        // Buat object AktivitasBerkurva
        AktivitasBerkurva ujianNilaiKurva = new AktivitasBerkurva(persenKurva);

        // Tetapkan skor ujian
        ujianNilaiKurva.setSkor(skor);

        // Tampilkan nilai mentah
        System.out.println("Nilai mentah = " +
                           ujianNilaiKurva.getNilaiMentah() +
                           " poin.");

        // Tampilkan nilai kurva
        System.out.println("Nilai kurva = " +
                           ujianNilaiKurva.getSkor());

        // Tampilkan grade huruf ujian
        System.out.println("Grade ujian = " +
                           ujianNilaiKurva.getGrade());
    }
}
```

Output Program (DemoAktivitasBerkurva.java)

```
Masukkan skor mentah siswa: 87
Masukkan persentase kurva: 1.06
Nilai mentah = 87.0 poin.
Nilai kurva = 92.22
Grade ujian = A
```

Program di atas menggunakan variabel `ujianNilaiKurva` untuk mereferensikan sebuah object `AktivitasBerkurva`. Pada baris 29, statement berikut digunakan untuk memanggil method `setSkor`:

```
ujianNilaiKurva.setSkor(skor);
```

Karena `ujianNilaiKurva` mereferensikan sebuah object `AktivitasBerkurva`, statement ini memanggil method `setSkor` dari class `AktivitasBerkurva`, bukan method versi superclassnya.

Perbedaan Overloading dengan Overriding

Overloading dan Overriding adalah dua hal berbeda. Pada topik sebelumnya kita membahas overloading. Overloading adalah ketika lebih dari satu method mempunyai nama yang sama tetapi dengan daftar parameter yang berbeda. Method ter-overloading mempunyai nama yang sama tetapi mempunyai signature yang berbeda. Sedangkan overriding adalah ketika method subclass mempunyai signature yang sama dengan method superclass.

Kita sebelumnya telah melihat method-method ter-overloading dapat berada di dalam sebuah class. Method overloading juga dapat digunakan dalam relasi inheritance. sebuah method di subclass dapat meng-overloading method di superclass. Namun, method overriding hanya dapat digunakan pada relasi inheritance. Kita tidak dapat meng-overriding method lain dalam class yang sama. Berikut adalah ringkasan perbedaan-perbedaan overloading dan overriding:

- Jika dua method memiliki nama sama namun signature yang berbeda, mereka disebut ter-overloading. Method overloading dapat dilakukan dalam class yang sama atau untuk method yang berada di superclass dan method lain yang berada di dalam subclass.
- Jika sebuah method dalam subclass mempunyai signature yang sama seperti sebuah method dalam superclass, method subclass ini disebut meng-overriding method superclass.

Perbedaan antara overloading dan overriding ini penting untuk diketahui. Ketika sebuah method dalam subclass meng-overloading method dalam superclass, kedua method tersebut dapat dipanggil dengan object subclass. Namun, ketika method dalam subclass meng-overriding method dalam superclass, hanya method versi subclass yang dapat dipanggil dengan object subclass. Sebagai contoh, perhatikan kode class `SuperClass3` berikut:

Definisi Class `SuperClass3` (SuperClass3.java)

```
public class SuperClass3
{
    /*
        Method ini menampilkan sebuah int.
        @param arg Sebuah int.
    */
    public void tampilkanNilai(int arg)
    {
        System.out.println("SUPERCLASS: " +
                           "Argument int adalah " + arg);
    }

    /*
        Method ini menampilkan sebuah String.
        @param arg Sebuah String.
    */
    public void tampilkanNilai(String arg)
    {
        System.out.println("SUPERCLASS: " +
                           "Argument String adalah " + arg);
    }
}
```

Class `SuperClass3` di atas mempunyai method ter-overloading bernama `tampilkanNilai`. Salah satu method `tampilkanNilai` menerima sebuah argument `int` dan method `tampilkanNilai` lainnya menerima sebuah argument `String`.

Sekarang perhatikan kode class `SubClass3` berikut yang mengekstensi class `SuperClass3`:

Definisi Class `SubClass3` (SubClass3.java)

```
public class SubClass3 extends SuperClass3
{
    /*
        Method ini meng-overriding sebuah method dalam superclass.
        @param arg Sebuah int.
    */
    @Override
    public void tampilkanNilai(int arg)
    {
        System.out.println("SUBCLASS: " +
                           "Argument int adalah " + arg);
    }

    /*
        Method ini meng-overloading method superclass.
        @param arg Sebuah doble.
    */
    public void tampilkanNilai(double arg)
    {
        System.out.println("SUBCLASS: " +
                           "Argument double adalah " + arg);
    }
}
```

Perhatikan bahwa `SubClass3` juga mempunyai dua method bernama `tampilkanNilai`. Method yang pertama, pada baris 8 sampai dengan 12, menerima sebuah argument `int`. Method ini meng-overriding salah satu method dalam superclass karena keduanya mempunyai signature yang sama. Method yang kedua, pada baris 18 sampai dengan 22, menerima sebuah argument `double`. Method ini meng-overloading method `tampilkanNilai` karena mempunyai signature yang berbeda dengan method `tampilkanNilai` lainnya. Meskipun sekarang terdapat total empat method `tampilkanNilai` dalam subclass `SubClass3` dan superclass `SuperClass33`, hanya tiga method yang dapat dipanggil dari object `SubClass3`. Program berikut mendemonstrasikan ini:

Program (DemoTampilkanNilai.java)

```
/*
    Program ini mendemonstrasikan method-method dalam
    class SuperClass3 dan SubClass3.
*/
public class DemoTampilkanNilai
{
    public static void main(String[] args)
    {
        // Buat object SubClass3
        SubClass3 myObject = new SubClass3();

        myObject.tampilkanNilai(10);        // Berikan sebuah int.
        myObject.tampilkanNilai(1.2);       // Berikan sebuah double.
        myObject.tampilkanNilai("Halo");    // Berikan sebuah String.
    }
}
```

Output Program (DemoTampilkanNilai.java)

```
SUBCLASS: Argument int adalah 10
SUBCLASS: Argument double adalah 1.2
SUPERCLASS: Argument String adalah Halo
```

Ketika argument `int` diberikan ke pemanggilan method `tampilkanNilai`, method yang berada dalam subclass yang dipanggil karena method ini meng-overriding method dalam superclass. Untuk memanggil method superclass yang di-overriding, kita harus menggunakan keyword `super` pada method di subclass. Berikut adalah contohnya:

```
public void tampilkanNilai(int arg)
{
    super.tampilkanNilai(arg);    // Memanggil method superclass.
    System.out.println("SUBCLASS: Argument int adalah " + arg);
}
```

Mencegah Method Di-overriding

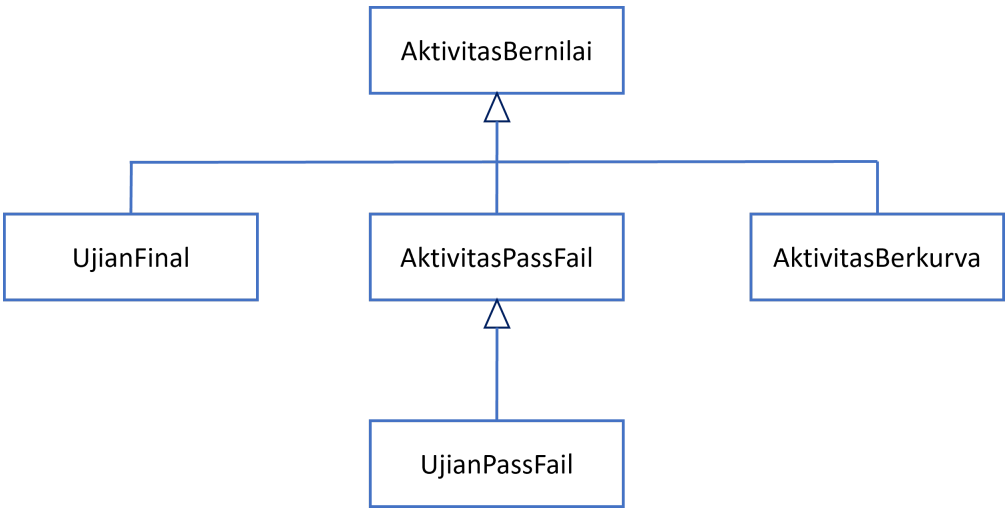
Kita dapat membuat sebuah method tidak dapat di-overriding oleh method pada subclass yang mengeksetensinya. Kita melakukannya dengan menambahkan modifier `final` pada header method tersebut. Berikut adalah contoh dari header method dengan modifier `final`:

```
public final void message()
```

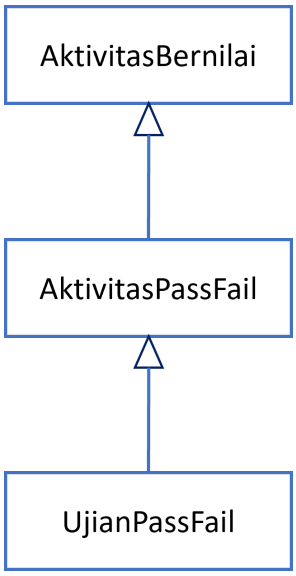
Jika sebuah subclass mencoba meng-overriding method `final`, compiler akan memberikan pesan error. Teknik ini digunakan untuk memastikan bahwa method superclass tertentu harus digunakan oleh subclass-subclassnya dan bukan versi modifikasi dari method tersebut.

9.4 Hirarki Inheritance

Sebuah class dapat diwarisi oleh lebih dari satu subclass. Selain itu, inheritance tidak hanya terbatas pada satu tingkat saja. Kita dapat menulis sebuah class yang mengekstensi class lain yang mengekstensi class lain juga. Kumpulan dari semua class-class yang mengekstensi sebuah superclass disebut sebagai hirarki inheritance. Gambar berikut mengilustrasikan sebuah hirarki inheritance:



Pada contoh-contoh sebelumnya kita teleh melihat bahwa kita dapat mempunyai lebih dari satu subclass dari class `AktivitasBernilai`. Class `UjianFinal` dan class `AktivitasBerkurva` keduanya mengekstensi class `AktivitasBernilai`. Sekarang kita akan melihat contoh class yang mengekstensi subclass dari class `AktivitasBernilai`. Kita akan membuat class `AktivitasPassFail` yang mengekstensi class `AktivitasBernilai` dan class `UjianPassFail` yang mengekstensi class `AktivitasPassFail`. Gambar berikut adalah diagram UML yang menunjukkan relasi dari dua class yang akan kita buat dengan class `AktivitasBernilai` :



Class `AktivitasPassFail` merepresentasikan aktivitas yang hanya memiliki dua grade huruf: "P" yang berarti pass (lulus) dan "F" yang berarti fail (gagal). Grade huruf dari `AktivitasPassFail` ini ditentukan berdasarkan apakah skor dari aktivitas ini melebihi skor kelulusan minimum. Berikut adalah kode untuk definisi class `AktivitasPassFail`:

Definisi Class `AktivitasPassFail` (AktivitasPassFail.java)

```
/*
    Class ini menyimpan skor numerik dan menentukan
    apakah skor tersebut pass (lulus) atau fail (gagal).
    Class ini mengekstensi class AktivitasBernilai.
*/
public class AktivitasPassFail extends AktivitasBernilai
{
    private double skorMinLulus;    // Skor minimum lulus

    /*
        Constructor menerapkan skor minimum lulus.
        @param sml Skor minimum lulus.
    */
    public AktivitasPassFail(double sml)
    {
        skorMinLulus = sml;
    }

    /*
        Method getGrade mengembalikan grade huruf
        yang ditentukan dari field skor. Method ini
        meng-overriding method superclass.
```

```
        @return Nilai huruf P atau F.
    */
    @Override
    public char getGrade()
    {
        char gradeHuruf;

        if (super.getSkor() >= skorMinLulus)
        {
            gradeHuruf = 'P';
        }
        else
        {
            gradeHuruf = 'F';
        }

        return gradeHuruf;
    }
}
```

Perhatikan kode class `AktivitasPassFail` di atas. Constructor dari class `AktivitasPassFail` yang dituliskan pada baris 14 sampai dengan 17, menerima sebuah argument tipe `double` yang merupakan nilai minimum lulus untuk aktivitas. Nilai ini disimpan dalam field `skorMinLulus`. Method `getGrade`, pada baris 26 sampai dengan 40, meng-overriding method superclass dengan nama sama. Method ini mengembalikan grade 'P' jika skor numerik lebih besar atau sama dengan `skorMinLulus` dan mengembalikan grade 'F' jika tidak.

Sekarang, misalkan kita ingin mengekstensi class `AktivitasPassFail` ke class lain yang lebih terspesialisasi. Sebagai contoh, kita membuat class `UjianPassFail` yang merepresentasikan sebuah ujian dengan grade 'P' yang menandakan lulus atau 'F' yang menandakan gagal. Class `UjianPassFail` mempunyai field-field: `banyakSoal` untuk banyaknya soal dalam ujian, `poinSetiapSoal` untuk besaran poin dari setiap soal, dan `banyakSalah` untuk banyaknya soal yang dijawal salah oleh siswa. Berikut adalah class `UjianPassFail`:

Definisi Class `UjianPassFail` (UjianPassFail.java)

```
/*
    Class ini menentukan grade pass (lulus) atau fail (gagal)
    dari sebuah ujian.
    Class ini mengekstensi class AktivitasPassFail.
*/
public class UjianPassFail extends AktivitasPassFail
{
    private int banyakSoal;         // Banyak soal dalam ujian final
    private double poinSetiapSoal;  // Poin untuk setiap soal
    private int banyakSalah;        // Banyak soal yang dijawab salah

    /*
        Constructor menetapkan nilai pada field banyak soal,
        banyak soal dijawab salah, dan skor minimum kelulusan.
        @param soal Banyak soal dalam ujian final.
        @param salah Bayak soal yang dijawab salah.
        @param minLulus Skor minimum lulus.
    */
    public UjianPassFail(int soal, int salah, double minLulus)
    {
```

```
        // Panggil constructor superclass
        super(minLulus);

        // Deklarasikan sbeuah lokal variabel untuk menyimpan skor.
        double skorNumerik;

        // Tetapkan field banyakSoal dan field banyakSalah.
        banyakSoal = soal;
        banyakSalah = salah;

        // Hitung point untuk setiap soal dan skor numerik
        // untuk ujian ini.
        poinSetiapSoal = 100.0 / soal;
        skorNumerik = 100.0 - (salah * poinSetiapSoal);

        // Panggil method setSkor superclass untuk
        // menetapkan skor numerik.
        setSkor(skorNumerik);
    }

    /*
        Method getPoinSetiapSoal mengembalikan poin untuk setiap saoal.
        @return Nilai dalam field poinSetiapSoal.
    */
    public double getPoinSetiapSoal()
    {
        return poinSetiapSoal;
    }

    /*
        Method getBanyakSalah mengembaliakan banyak soal
        yang dijawab salah.
        @return Nilai dalam field banyakSalah.
    */
    public int getBanyakSalah()
    {
        return banyakSalah;
    }
}
```

Class `UjianPassFail` mewarisi member-member dari class `AktivitasPassFail`, termasuk member-member yang diwarisi oleh class `AktivitasPassFail` dari class `AktivitasBernilai`. Program berikut mendemonstrasikan class `UjianPassFinal`:

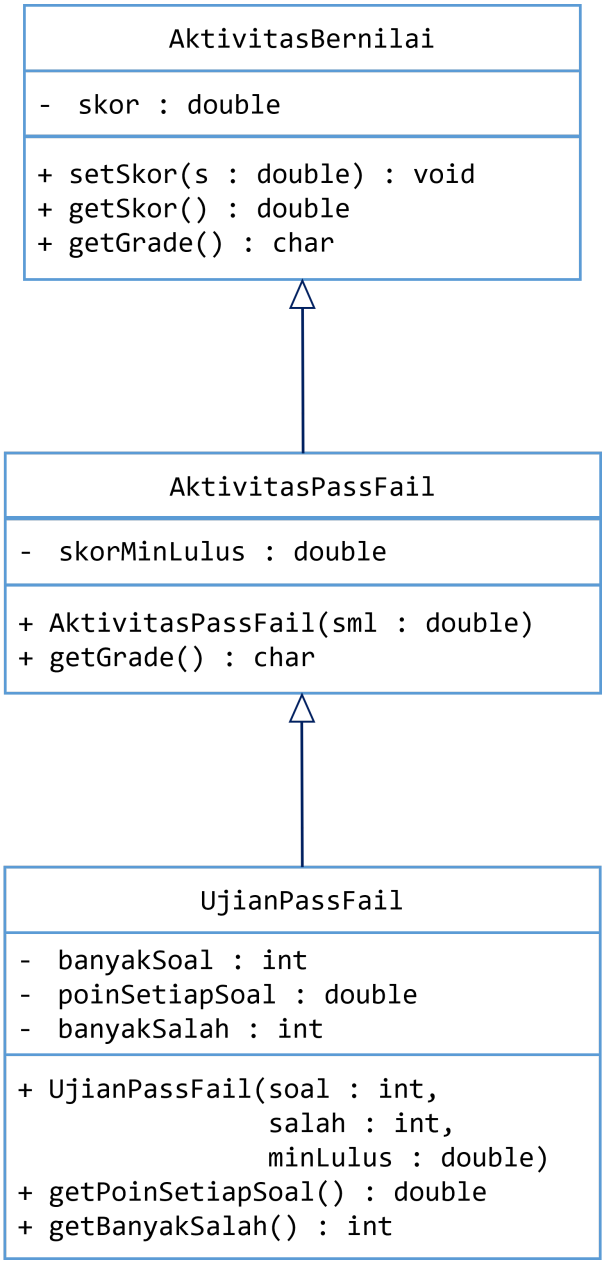
Program (DemoUjianPassFail.java)

```
import java.util.Scanner;

/*
    Program ini mendemonstrasikan class UjianPassFail.
*/
public class DemoUjianPassFail
{
    public static void main(String[] args)
    {
        int soal;           // Banyak soal.
        int salah;          // Banyak soal yang dijawab salah.
        double minLulus;    // Skor minimum lulus.

        // Buat object Scanner untuk menerima input keyboard.
        Scanner keyboard = new Scanner(System.in);

        // Dapatkan banyaknya soal dalam ujian.
        System.out.print("Berapa banyak soal dalam ujian ini: ");
        soal = keyboard.nextInt();

        // Dapatkan banyaknya soal yang dijawab salah.
         System.out.print("Berapa banyak soal yang dijawab salah oleh siswa ini:
");
        salah = keyboard.nextInt();

        // Dapatkan skor minimum lulus.
        System.out.print("Berapa skor minimum lulus dari ujian ini: ");
        minLulus = keyboard.nextInt();

        // Buat object UjianPassFail.
        UjianPassFail ujian = new UjianPassFail(soal, salah, minLulus);

        // Tampilkan poin untuk setiap soal.
        System.out.println("Poin setiap soal adalah " +
                           ujian.getPoinSetiapSoal() + " poin.");

        // Tampilkan skor ujian.
        System.out.println("Skor ujian siswa ini adalah " +
                           ujian.getSkor());

        // Tampilkan grade exam.
        System.out.println("Grade dari siswa ini adalah " +
                           ujian.getGrade());
    }
}
```

Output Program (DemoUjianPassFail.java)

```
Berapa banyak soal dalam ujian ini: 100
Berapa banyak soal yang dijawab salah oleh siswa ini: 25
Berapa skor minimum lulus dari ujian ini: 60
Poin setiap soal adalah 1.0 poin.
Skor ujian siswa ini adalah 75.0
Grade dari siswa ini adalah P
```

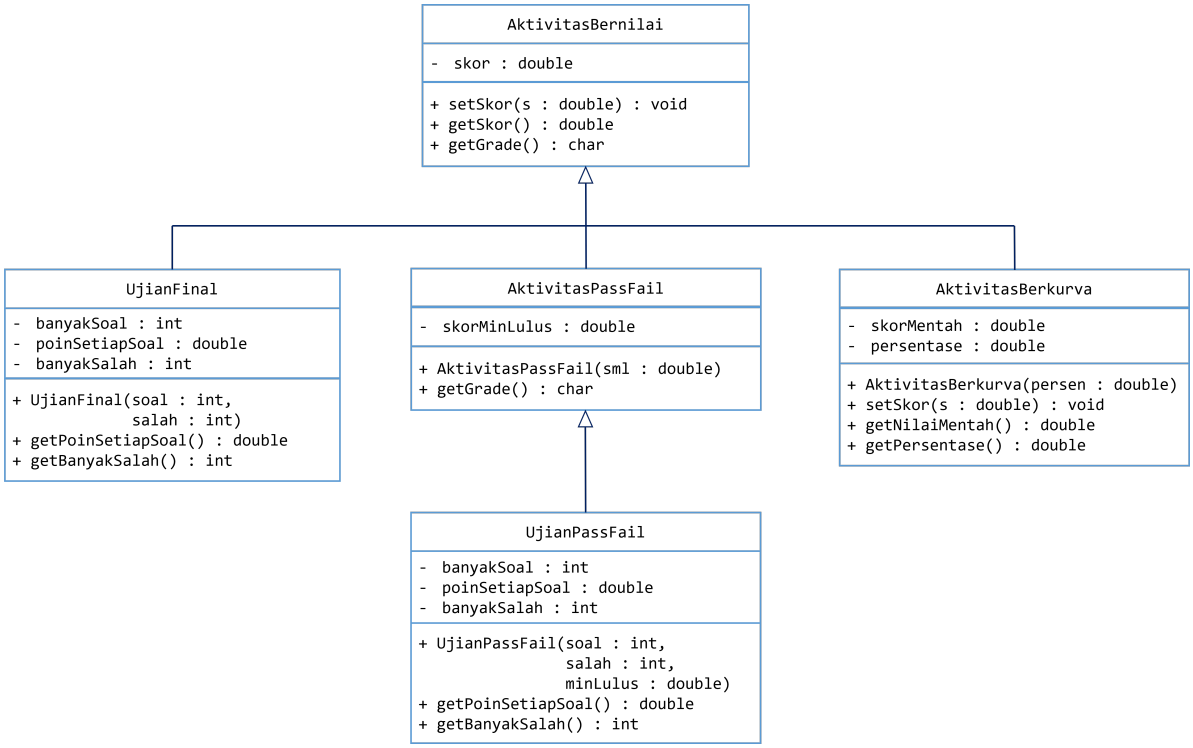
Gambar berikut adalah diagram UML yang menujukkan relasi inheritance antara class-class `AktivitasBernilai`, `AktivitasPassFail`, dan `UjianPassFail`.



9.5 Polymorphism

Polymorphism berarti banyak bentuk. Kata ini merupakan bentukan dua kata: poly yang berarti banyak dan moprh yang berarti bentuk. Dalam Java, variabel referensi bersifat polymorphic. Ini berarti variabel referensi mempunyai banyak bentuk. Maksud banyak bentuk dari variabel referensi adalah variabel referensi yang dideklarasikan dengan tipe suatu class selain dapat digunakan untuk mereferensikan sebuah object dari class itu sendiri juga dapat digunakan untuk mereferensikan object-object lain dari subclass-subclassnya.

Kita akan melihat polymorphism pada class `AktivitasBernilai` dan subclass-subclassnya. Sebagai pengingat, berikut adalah diagram UML dari class `AktivitasBernilai` dengan subclass-subclassnya.



Perhatikan kode berikut:

```
AktivitasBernilai ujian;
ujian = new UjianFinal(50, 7);
```

Statement pertama dari kode di atas adalah deklarasi variabel bertipe data `AktivitasBernilai`. Pada statement kedua kita membuat sebuah object dari class `UjianFinal` dan mereferensikannya ke variabel `ujian`. Kode di atas valid dan tidak menghasilkan error. Ini adalah contoh polymorphism dari variabel referensi. Sifat polymorphic dari variabel referensi memungkinkan sebuah variabel referensi dideklarasikan dengan tipe data dari suatu class dan menggunakannya untuk mereferensikan object dari subclass-subclass dari class tersebut.

Deklarasi-deklarasi variabel berikut juga valid karena class-class `UjianFinal`, `AktivitasPassFail`, dan `UjianPassFail` semuanya mewarisi class `AktivitasBernilai`:

```
AktivitasBernilai ujian1 = new UjianFinal(50, 7);
AktivitasBernilai ujian2 = new AktivitasPassFail(70);
AktivitasBernilai ujian3 = new UjianPassFail(100, 10, 70);
```

Perlu diperhatikan juga, meskipun variabel bertipe data superclass dapat mereferensikan object-object dari subclass-subclassnya, terdapat limitasi apa yang dapat dilakukan oleh variabel ini terhadap object-object tersebut. Variabel bertipe data superclass hanya dapat memanggil method-method yang dimiliki superclassnya meskipun variabel tersebut mereferensikan object dari subclass. Dengan kata lain, method-method dari subclass tidak dapat dipanggil melalui variabel tersebut.

Sebagai contoh, class `AktivitasBernilai` memiliki tiga method: `setSkor`, `getSkor`, dan `getGrade`. Misalkan kita membuat variabel bertipe data `AktivitasBernilai` dan menggunakannya untuk mereferensikan sebuah object dari calss `UjianPassFail`, variabel tersebut hanya dapat memanggil method `setSkor`, `getSkor`, dan `getGrade`. Meskipun, class `UjianPassFail` memiliki method `getPoinSetiapSoal`, method tersebut tidak dapat dipanggil melalui variabel tersebut. Perhatikan kode berikut:

```
AktivitasBernilai ujian = new UjianPassFail(100, 10, 7);
System.out.println(ujian.getSkor());              // Valid.
System.out.println(ujian.getGrade());             // Valid.
System.out.println(ujian.getPoinSetiapSoal());    // ERROR.
```

Pada kode di atas, variabel `ujian` bertipe data `AktivitasBernilai` dan mereferensikan sebuah object `UjianPassFail`. Karena `AktivitasBernilai` hanya memliki tidak method `setSkor`, `getSkor`, dan `getGrade`, kita hanya dapat memanggil ketiga method tersebut. Sehingga, statement kedua dan ketiga adalah statement yang valid. Statement terakhir mencoba memanggil method `getPoinSetiapSoal` yang hanya dimiliki oleh class `UjianPassFail`. Statement ini tidak valid karena variabel `ujian` hanya mengetahui mengenai method-method dalam class `AktivitasBernilai` sehingga variabel tersebut tidak dapat mengeksekusi method yang tidak dimiliki oleh class `AktivitasBernilai`.

Polymorphism dan Method yang Di-overriding

Jika sebuah variabel referensi yang dideklarasikan dengan tipe data superclass dan mereferensikan object dari subclassnya, apa yang terjadi jika subclass tersebut memiliki sebuah method yang meng-overriding sebuah method dalam superclass? Apakah variabel tersebut memanggil method versi superclassnya atau versi subclassnya? Sebagai contoh, perhatikan kode berikut:

```
AktivitasBernilai ujian = new AktivitasPassFail(60);
ujian.setSkor(70);
System.out.println(ujian.getGrade());
```

Ingat, class `AktivitasPassFail` mengekstensi class `AktivitasBernilai` dan meng-overriding method `getGrade`. Saat statement terakhir memanggil method `getGrade`, apakah statement tersebut memanggil method `getGrade` versi `AktivitasBernilai` (yang mengembalikan 'A', 'B', 'C', 'D', atau 'E') atau statement tersebut memanggil method `getGrade` versi `AktivitasPassFail` (yang mengembalikan 'P' atau 'F')? Jawaban dari pertanyaan ini adalah statement terakhir dari kode di atas akan memanggil method `getGrade` versi `AktivitasPassFail`. Sehingga statement terakhir akan memberikan output `P`.

Java menentukan method versi mana yang dipanggil dari variabel yang polymorphic dengan melakukan dynamic binding atau late binding. Dynamic binding (late binding) berarti penentuan method versi mana yang dipanggil dilakukan saat runtime (program berjalan). Dan karena saat runtime telah terbuat object yang direferensikan dalam memori, maka method versi dari object

yang direferensikan yang dipanggil.

Program berikut mendemonstrasikan perilaku polymorphic:

Program (Polymorphic.java)

```
/*
    Program ini mendemonstrasikan sifat polymorphic.
*/
public class Polymorphic
{
    public static void main(String[] args)
    {
        // Buat sebuah array bertipe AktivitasBernilai.
        AktivitasBernilai[] ujian = new AktivitasBernilai[3];

        // Ujian 1 adalah ujian bernilai dengan skor 95.
        ujian[0] = new AktivitasBernilai();
        ujian[0].setSkor(95);

        // Ujian 2 adalah ujian pass/fail. Siswa salah 5
        // dari 20 soal, dan skor minimum lulus adalah 60.
        ujian[1] = new UjianPassFail(20, 5, 60);

        // Ujian 3 adalah ujian final. Jumlah soal adalah 50
        // dan siswa salah 7.
        ujian[2] = new UjianFinal(50, 7);

        // Tampilkan nilai-nilai ujian.
        for (int i = 0; i < ujian.length; i++)
        {
            System.out.println("Ujian " + (i + 1) + ": " +
                               "\nSkor " + ujian[i].getSkor() +
                               "\nGrade " + ujian[i].getGrade());
            System.out.println();
        }
    }
}
```

Output Program (Polymorphic.java)

```
Ujian 1:
Skor 95.0
Grade A

Ujian 2:
Skor 75.0
Grade P

Ujian 3:
Skor 86.0
Grade B
```

Pada baris 9 program di atas, kita menuliskan statement berikut:

```
AktivitasBernilai[] ujian = new AktivitasBernilai[3];
```

Statement ini membuat sebuah array bernama `ujian` dengan tiga elemen yang setiap elemennya bertipe data `AktivitasBernilai`. Perhatikan pada ruas kanan operator assignment kita mempunyai ekspresi seperti berikut:

```
new AktivitasBernilai[3]
```

Ekspresi di atas bukanlah ekspresi instansiasi class `AktivitasBernilai`. Namun, ekspresi di atas membuat sebuah object array yang elemen-elemennya bertipe data `AktivitasBernilai`.

Pada baris 12, kita membuat sebuah object dari class `AktivitasBernilai` dan mereferensikannya ke elemen-0 dari array `ujian`. Lalu, pada baris 13, kita memanggil method `setSkor` untuk menetapkan nilai pada field `skor` dari object tersebut.

Pada baris 17, kita membuat sebuah object dari class `UjianPassFail` dan mereferensikan object tersebut ke elemen-1 dari array `ujian`. Dan pada baris 21 kita membuat sebuah object dari class `UjianFinal` dan mereferensikan object tersebut ke elemen-2 dari array `ujian`.

Pada baris 24 sampai dengan 30, kita menuliskan sebuah loop `for` untuk mengiterasi elemen-elemen array `ujian`. Di dalam body loop `for`, pada baris 26 sampai dengan 28, kita menuliskan statement `println` yang menampilkan hasil-hasil pemanggilan method `getSkor` dan `getGrade`. Kita dapat memanggil dua method tersebut karena kedua method ini didefinisikan pada class `AktivitasBernilai`. Pada pemanggilan method `getGrade`, karena method ini di-overriding dalam class `UjianPassFail` maka pemanggilan method `getGrade` elemen-1 akan memanggil method `getGrade` versi class `UjianPassFail`.

Operator `instanceof`

Terdapat sebuah operator bernama `instanceof` yang dapat kita gunakan untuk menguji apakah sebuah object adalah sebuah instance dari suatu class tertentu. Berikut adalah format umum ekspresi yang menggunakan operator `instanceof`:

```
varRef instanceof NamaClass
```

Ekspresi dengan operator `instanceof` di atas adalah ekspresi Boolean yang mengembalikan `true` jika object yang direferensikan oleh `varRef` adalah sebuah instance dari `NamaClass`. Jika object yang direferensikan oleh `varRef` bukanlah sebuah instance dari `NamaClass`, ekspresi ini akan dievaluasi ke `false`.

Sebagai contoh, statement `if` pada kode berikut menguji apakah variabel referensi `boks` mereferensikan object `PersegiPanjang`:

```
PersegiPanjang boks = new PersegiPanjang();
if (boks instanceof PersegiPanjang)
{
    System.out.println("Ya, boks adalah PersegiPanjang.");
}
else
{
    System.out.println("Tidak, boks bukan PersegiPanjang.");
}
```

Kode di atas akan menampilkan "Ya, boks adalah PersegiPanjang" karena object yang direferensikan oleh variabel `boks` adalah instance dari class `PersegiPanjang`.

Operator `instanceof` juga akan mengembalikan `true` jika variabel referensi diuji terhadap superclassnya. Sebagai contoh, perhatikan kode berikut:

```
Balok kotak = new Balok(12.0, 6.0, 5.0);
if (kotak instanceof PersegiPanjang)
{
    System.out.println("Ya, kotak adalah PersegiPanjang.");
}
else
{
    System.out.println("Tidak, kotak bukan PersegiPanjang.");
}
```

Meskipun object yang direferensikan oleh `kotak` adlaah object `Balok`, kode di atas akan menampilkan "Ya, kotak adalah PersegiPanjang." Ini karena `Balok` adalah subclass dari `PersegiPanjang`.

Relasi "Is-a" Tidak Bekerja Dalam Kebalikannya

Perlu dicatat polymorphism hanya bekerja jika variabel referensi dideklarasikan dengan tipe superclass dan digunakan untuk merefrensikan subclass-nya. Polymorphism tidak bekerja dalam kebalikannya. Kita tidak dapat mempunyai variabel referensi yang dideklarasikan dengan tipe subclass lalu digunakan untuk mereferensikan superclass-nya. Sebagai contoh, kode berikut akan menghasilkan error kompilasi:

```
UjianFinal ujian;
ujian = new AktivitasBernilai();    // ERROR!
```

Pada kode kita mendeklarasikan variabel `ujian` dengan tipe `UjianFinal`, lalu menugaskan variabel tersebut untuk mereferensikan object dari class `AktivitasBernilai` yang merupakan superclass dari class `UjianFinal`. Kode di atas akan menghasilkan error.

Kode di atas menghasilkan error karena polymorphism hanya bekerja satu arah. Alasan kenapa polymorphism hanya bekerja satu arah adalah karena polymorphism bekerja berdasarkan relasi "is-a". Ingat, object dari class yang mengekstensi suatu superclass mempunyai relasi "is-a" dengan object superclassnya. Ini berarti object `UjianFinal` "is-a" (adalah sebuah) object `AktivitasBernilai`. Namun, kebalikannya tidak benar. Tidak semua object `AktivitasBernilai` adalah object `UjianFinal`.

Casting

Seperti yang telah disebutkan sebelumnya bahwa terdapat limitasi pada variabel bertipe data superclass yang digunakan untuk mereferensikan object-object dari subclass-subclassnya. Variabel tersebut hanya dapat memanggil method yang dimiliki oleh superclass-nya. Bagaimana jika kita ingin memanggil method-method yang hanya dimiliki subclass-nya? Kita dapat melakukannya dengan terlebih dahulu melakukan casting tipe data ke subclass tersebut. Sebagai contoh, perhatikan kode berikut:

```
AktivitasBernilai ujian = new AktivitasPassFail();
AktivitasPassFail kuis = (AktivitasPassFail) ujian;
```

Pada statement kedua dari kode di atas, operator casting `(AktivitasPassFail)` melakukan casting variabel `ujian` yang bertipe data `AktivitasBernilai` ke tipe data `UjianPassFail`. Setelah statement ini dieksekusi variabel `kuis` akan mereferensikan object dari class `AktivitasPassFail` yang direferensikan oleh variabel `ujian`. Dan karena sekarang variabel `kuis` bertipe data class sebenarnya dari object yang direferensikan maka variabel `kuis` mengetahui method-method apa saja yang dimiliki oleh object tersebut.

Terdapat hal yang perlu diperhatikan saat kita melakukan casting tipe object: kita harus memastikan class tujuan casting adalah class sebenarnya dari object yang ingin di-casting. Jika kita melakukan casting ke class yang bukan class dari object yang di-casting, kita akan mendapatkan error run-time. Kita dapat memastikan class tujuan casting adalah class sebenarnya dari object yang ingin kita casting dengan melakukan pengujian menggunakan operator `instanceof`, seperti dapat dilihat pada kode berikut:

```
if (ujian instanceof AktivitasPassFail)
{
    AktivitasPassFail kuis = (AktivitasPassFail) ujian;
    // Sekarang kita dapat memanggil method dari class AktivitasPassFail
    // melalui variabel kuis.
}
```

Program berikut, modifikasi dari program `Polymorphic.java` sebelumnya, mendemonstrasikan penggunaan casting antar object:

Program (Polymorphic2.java)

```
/*
    Program ini mendemonstrasikan casting object.
*/
public class Polymorphic2
{
    public static void main(String[] args)
    {
        // Buat sebuah array bertipe AktivitasBernilai.
        AktivitasBernilai[] ujian = new AktivitasBernilai[3];

        // Ujian 1 adalah ujian bernilai dengan skor 95.
        ujian[0] = new AktivitasBernilai();
        ujian[0].setSkor(95);

        // Ujian 2 adalah ujian pass/fail. Siswa salah 5
        // dari 20 soal, dan skor minimum lulus adalah 60.
        ujian[1] = new UjianPassFail(20, 5, 60);

        // Ujian 3 adalah ujian final. Jumlah soal adalah 50
        // dan siswa salah 7.
        ujian[2] = new UjianFinal(50, 7);


        // Tampilkan nilai-nilai ujian.
        for (int i = 0; i < ujian.length; i++)
        {
            System.out.println("Ujian " + (i + 1) + ": " +
                               "\nSkor " + ujian[i].getSkor() +
                               "\nGrade " + ujian[i].getGrade());

            if (ujian[i] instanceof UjianFinal)
            {
                UjianFinal finalExam = (UjianFinal) ujian[i];
                System.out.println("Poin setiap soal = " +
                                   finalExam.getPoinSetiapSoal());
            }

            System.out.println();
        }
    }
}
```

Output Program (Polymorphic2.java)

```
Ujian 1:
Skor 95.0
Grade A

Ujian 2:
Skor 75.0
Grade P

Ujian 3:
Skor 86.0
Grade B
Poin setiap soal = 2.0
```

Pada program di atas, di baris 31 sampai dengan 34, Kita menuliskan kode yang melakukan casting ke tipe data `UjianFinal` jika elemen array `ujian` adalah instance dari class `UjianFinal`. Setelah melakukan casting, kita memanggil method `getPoinSetiapSoal` yang dimiliki oleh class `UjianFinal` tetapi tidak dimiliki oleh class `AktivitasBernilai`.

9.6 Class `Object`

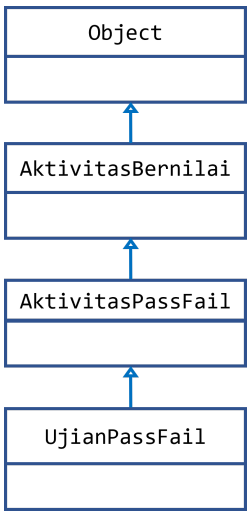
Semua class dalam Java secara langsung atau tidak langsung mewarisi dari sebuah class bernama `Object`. Class yang tidak ditulis dengan keyword `extends` untuk mewarisi dari class lain, secara otomatis di-ekstensi oleh Java untuk mewarisi dari class `Object`. Sebagai contoh, perhatikan definisi class berikut:

```
public class MyClass
{
    // Deklarasi Member-member...
}
```

Class di atas tidak secara eksplisit mengekstensi class lain, sehingga Java memperlakukan class ini seperti seolah class tersebut dituliskan seperti berikut:

```
public class MyClass extends Object
{
    // Deklarasi Member-member...
}
```

Semua class mewarisi dari class `Object`. Gambar berikut menunjukkan bagaimana hirarki inheritance dari class `UjianPassFail` terhadap class `Object`:



Karena semua class secara langsung atau tidak langsung mengekstensi class `Object`, maka semua class mewarisi member-member dari class `Object`. Class `Object` mempunyai method-method generik, dua diantaranya:

- Method `toString`: yang mengembalikan sebuah `String` berisi deksripsi object.
- Method `equals` : yang membandingkan object dengan object lain.

Kedua object di atas umumnya di-overriding oleh class-class. Method `toString` umumnya di-overriding untuk menampikan state instance dan method `equals` umumnya di-overriding untuk membandingkan apakah kedua object mempunyai nilai-nilai field yang sama.

Meng-overriding Method `toString`

Misalkan pada class `PersegiPanjang` kita dapat menuliskan method `toString` yang mengembalikan string yang berisi informasi mengenai state dari object seperti berikut:

```
public class PersegiPanjang
{
    // ... kode-kode lainnya tidak ditampilkan disini.

    public String toString()
    {
        String str = "Panjang: " + panjang +
                     "\nLebar: " + lebar;

        return str;
    }
}
```

Method `toString` dipanggil otomatis ketika kita mengkonkatenasi sebuah string dengan object. Sebagai contoh, perhatikan kode berikut:

```
PersegiPanjang boks = new PersegiPanjang(12.0, 5.0);
String s = "Informasi persegi panjang:\n" + boks;
System.out.println(s);
```

Kode di atas akan memberikan output berikut:

```
Informasi persegi panjang:
Panjang: 12.0
Lebar: 5.0
```

Method `toString` juga dipanggil otomatis ketika kita memberikan object ke method yang mencetak ke console (`print`, `println`, atau `printf`). Perhatikan kode berikut:

```
PersegiPanjang boks = new PersegiPanjang(12.0, 5.0);
System.out.println(boks);
```

Kode di atas akan memberikan output berikut:

```
Panjang: 12.0
Lebar: 5.0
```

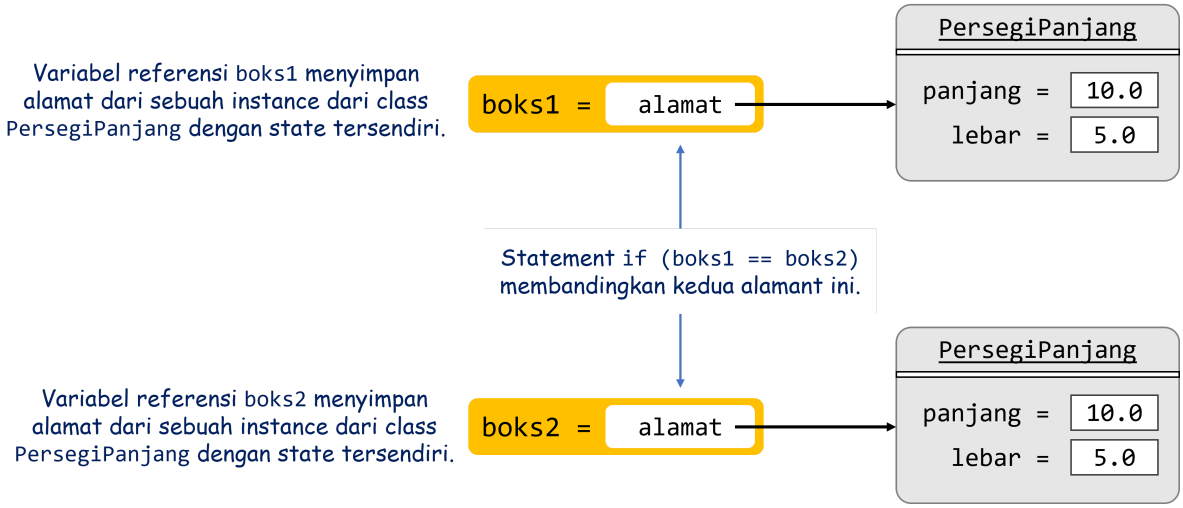
Meng-overriding Method `equals`

Method `equals` umumnya di-overriding untuk menguji apakah field-field dari dua object mempunyai nilai-nilai yang sama. Method ini bekerja berbeda dengan operator `==`. Operator `==` membandingkan apakah dua variabel mereferensikan object yang sama, sedangkan method `equals` digunakan untuk membandingkan apakah nilai-nilai field dari dua object adalah sama.

Kita tidak dapat menggunakan operator `==` untuk membandingkan nilai-nilai field dari dua object. Sebagai contoh, kode berikut terlihat seperti membandingkan isi dari dua object `PersegiPanjang`, tetapi sesungguhnya tidak:

```
PersegiPanjang boks1 = new PersegiPanjang(10.0, 5.0);
PersegiPanjang boks2 = new PersegiPanjang(10.0, 5.0);

if (boks1 == boks2)
{
    System.out.println("boks1 dan boks2 memiliki panjang dan lebar yang sama.");
}
else
{
        System.out.println("boks1 dan boks2 memiliki panjang dan lebar yang
berbeda.");
}
```

Kode di atas akan memberikan output: "boks1 dan boks2 memiliki panjang dan lebar yang berbeda". Ini karena operator `==` tidak membandingkan isi (nilai-nilai field) dari object `boks1` dan `boks2` tetapi membandingkan alamat memori yang disimpan oleh object `boks1` dan `boks2`. Karena `boks1` dan `boks2` mereferensikan dua object yang berbeda (meskipun mempunyai nilai-nilai field yang sama), maka ekspresi `boks1 == boks2` akan dievaluasi ke `false`.



Kita dapat meng-overriding method `equals` pada class `Object` sehingga kita dapat membandingkan isi dari dua object dari suatu class. Kita dapat meng-overriding method `equals` pada class `PersegiPanjang` dengan menambahkan definisi method `equals` seperti berikut:

Potongan Definisi Class `PersegiPanjang` (PersegiPanjang.java)

```
public class PersegiPanjang
{
    ...kode-kode yang ditulis pada bagian sebelumnya tidak ditampilkan

    public boolean equals(Object objectLain)
    {
        boolean status;

        // Kita menguji apakah argument object yang diberikan
        // adalah instance dari PersegiPanjang. Pengujian ini untuk
        // memastikan kita dapat melakukan casting.
        if (objectLain instanceof PersegiPanjang)
        {
            PersegiPanjang p = (PersegiPanjang) objectLain;
            if (this.panjang == p.panjang && this.lebar == p.lebar)
            {
                status = true;
            }
            else
            {
                status = false;
            }
        }
        else
        {
            status = false;
        }
        return status;
    }
}
```

Pada kode method `equals` di atas kita menggunakan variabel `status` untuk menyimpan nilai Boolean hasil pengujian. Pada baris 12 sampai dengan 27, kita menuliskan statement `if` tersarang. Statement `if-else` bagian luar menguji apakah `objectLain` yang diberikan sebagai argument adalah instance dari class `PersegiPanjang`. Jika `objectLain` yang diberikan bukanlah instance dari class `PersegiPanjang` maka sudah pasti `objectLain` tersebut tidak mempunyai nilai-nilai field yang sama dengan object ini. Sehingga kita menetapkan variabel `status` dengan `false` di dalam klausa `else`.

Pengujian apakah `objectLain` adalah isntance dari class `PersegiPanjang` diperlukan untuk memastikan casting yang kita lakukan pada baris 14 dapat dilakukan. Jika `objectLain` ini bukanlah instance dari class `PersegiPanjang` maka statement pada baris 14 akan menghasilkan error.

Lalu, pada statement `if-else` bagian dalam, baris 15 sampai dengan 22, kita menguji apakah field `panjang` dan field `lebar` dari object ini mempunyai nilai-nilai yang sama dengan field `panjang` dan field `lebar` dari `objectLain`. Jika ya kita menetapkan `status` dengan `true` dan jika tidak kita menetapkan `status` dengan `false`. Pada statement terakhir, baris 28, method ini mengembalikan nilai dalam variabel `status`.

Class `PersegiPanjang` dengan Method `toString` dan `equals`

Berikut adalah kode lengkap dari class `PersegiPanjang` dengan method `toString` dan method `equals` yang kita tambahkan sebelumnya:

Definisi Class `PersegiPanjang` (PersegiPanjang.java)

```
/*
    Class PersegiPanjang dengan method toString dan
    method equals.
*/
public class PersegiPanjang
{
    private double panjang;
    private double lebar;

    /*
        Constructor
        @param pjg Panjang dari persegi panjang.
        @param lbr Lebar dari persegi panjang.
    */
    public PersegiPanjang()
    {
        panjang = 0.0;
        lebar = 0.0;
    }

    /*
        Constructor
        @param pjg Panjang dari persegi panjang.
        @param lbr Lebar dari persegi panjang.
    */
    public PersegiPanjang(double panjang, double lebar)
    {
        this.panjang = panjang;
```

```
        this.lebar = lebar;
    }

    /*
        Constructor
        @param sisi Panjang dan lebar dari persegi panjang.
    */
    public PersegiPanjang(double sisi)
    {
        this(sisi, sisi);
    }

    /*
        Method setPanjang menyimpan sebuah nilai dalam field panjang.
        @param pjg Nilai yang disimpan dalam field panjang.
    */
    public void setPanjang(double panjang)
    {
        this.panjang = panjang;
    }

    /*
        Method setLebar menyimpan sebuah nilai dalam field lebar.
        @param lbr Nilai yang disimpan dalam field lebar.
    */
    public void setLebar(double lebar)
    {
        this.lebar = lebar;
    }

    /*
        Method getPanjang mengembalikan panjang dari object PersegiPanjang.
        @return Nilai dalam field panjang
    */
    public double getPanjang()
    {
        return panjang;
    }

    /*
        Method getLebar mengembalikan lebar dari object PersegiPanjang.
        @return Nilai dalam field lebar
    */
    public double getLebar()
    {
        return lebar;
    }

    /*
        Method getLuas mengembalikan luas dari object PersegiPanjang.
        @return Hasil dari panjang kali lebar.
    */
    public double getLuas()
    {
        return panjang * lebar;
    }

    /*
```

```
        Method toString mengembalikan sebuah string
        berisi informasi mengenai state dari object.
        @return String deskripsi state object.
    */
    @Override
    public String toString()
    {
        String str = "Panjang: " + panjang +
                     "\nLebar: " + lebar;

        return str;
    }

    /*
        Method equals mengembalikan Bolean true jika object argument
        memiliki nilai-nilai field yang sama.
        @param objectLain Object yang ingin dibandingkan.
        @return Boolean hasil uji nilai-nilai field.
    */
    @Override
    public boolean equals(Object objectLain)
    {
        boolean status;

        // Kita menguji apakah argument object yang diberikan
        // adalah instance dari PersegiPanjang. Pengujian ini untuk
        // memastikan kita dapat melakukan casting.
        if (objectLain instanceof PersegiPanjang)
        {
            PersegiPanjang p = (PersegiPanjang) objectLain;
            if (this.panjang == p.panjang && this.lebar == p.lebar)
            {
                status = true;
            }
            else
            {
                status = false;
            }
        }
        else
        {
            status = false;
        }
        return status;
    }
}
```

Program berikut mendemonstrasikan method `toString` dan `equals` dari class `PersegiPanjang`:

Program (DemoPersegiPanjang2.java)

```
public class DemoPersegiPanjang2
{
    public static void main(String[] args)
    {
        PersegiPanjang boks1 = new PersegiPanjang(12.0, 5.0);
        PersegiPanjang boks2 = new PersegiPanjang(12.0, 5.0);
        PersegiPanjang boks3 = new PersegiPanjang(20.0, 12.0);

        System.out.println("Boks1: ");
        System.out.println(boks1);
        System.out.println();
        System.out.println("Boks2: ");
        System.out.println(boks2);
        System.out.println();
        System.out.println("Boks3: ");
        System.out.println(boks3);
        System.out.println();

        if (boks1.equals(boks2))
        {
            System.out.println("boks1 sama dengan boks2.");
        }
        else
        {
            System.out.println("boks1 tidak sama dengan boks2.");
        }

        if (boks1.equals(boks3))
        {
            System.out.println("boks1 sama dengan boks3.");
        }
        else
        {
            System.out.println("boks1 tidak sama dengan boks3.");
        }
    }
}
```

Output Program (DemoPersegiPanjang2.java)

```
Boks1:
Panjang: 12.0
Lebar: 5.0

Boks2:
Panjang: 12.0
Lebar: 5.0

Boks3:
Panjang: 20.0
Lebar: 12.0

boks1 sama dengan boks2.
boks1 tidak sama dengan boks3.
```

9.7 Class Abstract dan Method Abstract

Class abstract adalah class yang tidak dapat diinstansiasi. Class ini ditujukan untuk diwarisi oleh class-class lain. Class abstract umumnya mempunyai setidaknya satu method abstract. Method abstract adalah method yang hanya memiliki header tanpa body. Berikut adalah syntax umum dari method abstract:

```
AccessSpecifier abstract TipeReturn NamaMethod(Parameter...);
```

Perhatikan bahwa terdapat keyword `abstract` pada header dari method abstract. Perhatikan juga bahwa header ini diakhiri dengan titik koma (semicolon) dan tidak ada body untuk method abstract ini. Berikut adalah contoh sebuah method abstract:

```
public abstract void setNilai(int nilai);
```

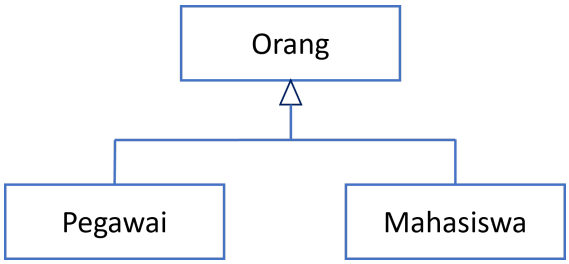
Jika terdapat method abstract di dalam sebuah class, method tersebut harus di-overriding oleh sebuah method dalam subclass yang mengekstensi class tersebut. Jika tidak terdapat method yang meng-overriding, maka compiler akan memberikan pesan error. Method abstract digunakan untuk memastikan subclass mengimplementasikan (menuliskan kode-kode untuk) method tersebut.

Jika terdapat sebuah class yang berisi sebuah method abstract, kita tidak dapat membuat instance dari class tersebut. Method abstract umumnya digunakan di dalam class abstract. Class abstract tidak untuk diinstansiasi, tetapi berfungsi sebagai superclass untuk class-class lain. Class abstract dapat dipandang sebagai class yang belum lengkap, dan subclass-subclass yang mengekstensinya bertugas untuk melengkapinya.

Class abstract didefinisikan dengan menuliskan keyword `abstract` pada headernya. Berikut adalah syntax umum header dari class abstract:

```
AccessSpecifier abstract class NamaClass
```

Untuk mencontohkan class abstract, misalkan kita mempunyai hirarki inherintance class `Orang` dan subclass-subclassnya seperti berikut:



Kita menginginkan class `Orang` berfungsi sebagai basis dari class-class yang merepresentasikan pegawai dan mahasiswa. Kita tidak menginginkan class `Orang` di-instansiasi langsung. Sehingga kita mendefinisikan class `Orang` sebagai class abstract. Berikut adalah class abstract `Orang`:

Definisi Class Abstract `Orang` (Orang.java)

```
/*
    Class Orang adalah class abstract yang menyimpan nama seseorang.
    Class-class yang mewakili tipe tertentu orang mewarisi dari class ini.
*/
public abstract class Orang
{
    private String nama;    // Nama orang

    /*
        Constructor menetapkan nama orang.
        @param nama Nama orang ini.
    */
    public Orang(String nama)
    {
        this.nama = nama;
    }

    /*
        Method getNama mengembalikan nama orang ini.
        @return Nilai field nama.
    */
    public String getNama()
    {
        return nama;
    }

    /*
        Method getDeskripsi adalah method abstract.
        Method ini harus di-overriding di dalam subclass.
        @return Deskripsi mengenai orang.
    */
    public abstract String getDeskripsi();
}
```

Class `Orang` memiliki field `nama` untuk menyimpan nama orang. Selain itu, class ini juga mempunyai sebuah constructor, method `getNama`, dan sebuah method abstract bernama `getDeskripsi`. Tujuan dari method `getDeskripsi` adalah mengembalikan sebuah `String` berupa deskripsi status dari orang, seperti:

```
seorang pegawai bekerja sebagai Programmer dengan gaji 3500000
seorang mahasiswa jurusan Teknik Informatika dengan ipk 3.8
```

Method abstract `getDeskripsi` harus di-overriding di dalam subclass yang mengekstensi class `Orang`. Method ini dibuat abstract karena class `Orang` hanya mengetahui nama orang saja dan informasi-informasi lainnya dimaksudkan untuk ditambahkan pada subclass-subclassnya.

Perhatikan juga pada class abstract `Orang` di atas, kita menuliskan method konkrit (non-abstract) bernama `getNama`. Pada class abstract, kita dapat mempunyai method abstract dan juga method konkrit. Kita bahkan dapat mendeklarasikan sebuah class sebagai abstract meskipun class tersebut tidak mempunyai method abstract.

Sekarang kita akan melihat class `Pegawai` dan class `Mahasiswa` yang mengekstensi class `Orang`. Class `Pegawai` dapat dituliskan seperti berikut:

Definisi Class `Pegawai` (Pegawai.java)

```
/*
    Class ini mengekstensi class abstract Orang
    untuk merepresentasikan pegawai.
*/
public class Pegawai extends Orang
{
    String pekerjaan;       // Pekerjaan pegawai
    private double gaji;    // Gaji pegawai

    /*
        Constructor
        @param nama Nama pegawai.
        @param pekerjaan Pekerjaan pegawai.
        @param gaji Gaji pegawai.
    */
    public Pegawai(String nama, String pekerjaan, double gaji)
    {
        super(nama);
        this.pekerjaan = pekerjaan;
        this.gaji = gaji;
    }

    /*
        Method getPekerjaan mengembalikan pekerjaan pegawai.
        @return Nilai pada field pekerjaan.
    */
    public String getPekerjaan()
    {
        return pekerjaan;
    }

    /*
        Method getGaji mengembalikan gaji pegawai.
        @return Nilai pada field gaji.
    */
    public double getGaji()
    {
        return gaji;
    }

    /*
        Method getDeskripsi mengembalikan string deskripsi pegawai.
        @return String deksripsi pegawai.
    */
    @Override
    public String getDeskripsi()
    {
        String deskripsi;
        deskripsi = "seorang pegawai bekerja sebagai " +
                    getPekerjaan() + " dengan gaji " + getGaji();

        return deskripsi;
    }
}
```

Class `Pegawai` yang mengekstensi class `Orang` mempunyai field-field berikut: `pekerjaan` untuk menyimpan pekerjaan pegawai ini dan `gaji` untuk menyimpan gaji dari pegawai ini. Method-method accessor, `getPekerjaan` dan `getGaji`, juga disediakan untuk mengakses kedua field tersebut. Selain itu, class `Pegawai` meng-overriding method abstract `getDeskripsi` untuk menampilkan deskripsi pegawai berdasarkan pekerjaannya dan gajinya.

Class `Mahasiswa` dapat dituliskan seperti berikut:

Definisi Class `Mahasiswa` (Mahasiswa.java)

```
/*
    Class ini mengekstensi class abstract Orang
    untuk merepresentasikan mahasiswa.
*/
public class Mahasiswa extends Orang
{
    private String jurusan;     // Jurusan mahasiswa
    private double ipk;         // IPK mahasiswa

    public Mahasiswa(String nama, String jurusan, double ipk)
    {
        super(nama);
        this.jurusan = jurusan;
        this.ipk = ipk;
    }

    /*
        Method getJurusan mengembalikan jurusan mahasiswa.
        @return Nilai field jurusan.
    */
    public String getJurusan()
    {
        return jurusan;
    }

    /*
        Method getIPK mengembalikan ipk mahasiswa.
        @return Nilai field ipk.
    */
    public double getIPK()
    {
        return ipk;
    }

    /*
        Method getDeskripsi mengembalikan string deskripsi mahasiswa.
        @return String deksripsi mahasiswa
    */
    @Override
    public String getDeskripsi()
    {
        String deskripsi;
        deskripsi = "seorang mahasiswa jurusan " +
                    getJurusan() + " dengan IPK " + getIPK();

        return deskripsi;
    }
```

```
}
```

Class `Mahasiswa` mengekstensi class abstract `Orang`. Class ini memiliki dua field: `jurusan` untuk menyimpan jurusan mahasiswa dan `ipk` untuk menyimpan ipk mahasiswa. Method-method accessor, `getJurusan` dan `getIPK`, juga disediakan untuk mengakses kedua field tersebut. Sama seperti dalam class `Pegawai`, di dalam class `Mahasiswa`, method `getDeskripsi` juga meng-overriding method `getDeskripsi` dari class `Orang`. Method `getDeskripsi` digunakan untuk menampilkan deskripsi mahasiswa berdasarkan jurusannya dan ipknya.

Sebelum kita melihat kode yang mendemonstrasikan class `Orang` mari kita bahas ketentuan-ketentuan dari class abstract. Hal paling penting yang perlu kita ingat adalah class abstract tidak dapat di-instansiasi. Dengan kata lain, jika sebuah class dideklarasikan sebagai `abstract`, kita tidak bisa membuat object-object dari class tersebut. Sebagai contoh, statement berikut:

```
Orang siswa = new Orang("Budi Susilo");    // ERROR.
```

akan menghasilkan error. Namun, kita dapat menggunakan class abstract sebagai tipe data dari variabel referensi, tetapi variabel referensi tersebut harus mereferensikan ke sebuah object dari subclass non-abtract-nya. Sebagai contoh, kita dapat mempunyai statement berikut:

```
Orang o = new Mahasiswa("Herman Budi", "Sistem Informasi", 3.5);
```

Pada statement di atas, variabel `o` adalah variabel referensi bertipe data `Orang` dan mereferensikan sebuah instance dari subclass `Mahasiswa`.

Berikut adalah program yang mendemonstrasikan penggunaan class `Pegawai` dan class `Mahasiswa`:

Program (DemoOrang.java)

```
/*
    Program ini mendemonstrasikan class Pegawai
    dan class Mahasiswa.
*/
public class DemoOrang
{
    public static void main(String[] args)
    {
        // Buat aray bertipe Orang dengan 2 elemen.
        Orang[] grup = new Orang[2];

        // Buat object Pegawai dan tugaskan ke elemen-0.
        grup[0] = new Pegawai("Budi Susilo", "Programmer", 3500000);

        // Buat object Mahasiswa dan tugaskan ke elemen-1.
        grup[1] = new Mahasiswa("Hari Sudiro", "Teknik Informatika", 3.8);

        // Iterasi array grup dan panggil method-method pada elemen
        // utnuk menampilkan deskripsi orang.
        for (Orang o : grup)
        {
            System.out.println(o.getNama() + " adalah " +
                               o.getDeskripsi());
        }
```

```
    }
}
```

Output Program (DemoOrang.java)

```
Budi Susilo adalah seorang pegawai bekerja sebagai Programmer dengan gaji
3500000.0
Hari Sudiro adalah seorang mahasiswa jurusan Teknik Informatika dengan IPK 3.8
```

Perhatikan program di atas, pada baris 10, kita mempunyai statement berikut:

```
Orang[] grup = new Orang[2];
```

Statement ini membuat sebuah array dengan dua elemen yang semua elemennya bertipe data `Orang`. Ini menunjukkan bahwa meskipun class abstract tidak dapat diinstansiasi, tetapi kita tetap dapat menggunakannya sebagai tipe data.

Lalu, pada baris 13, kita membuat object `Pegawai` dan menugaskan alamat object tersebut ke elemen-0 dari array `grup`. Dan pada baris 16, kita membuat object `Mahasiswa` dan menugaskan alamat object tersebut ke elemen-1 dari array `grup`. Ingat disini kita menggunakan sifat polymorphic dari elemen-elemen dari array `grup`.

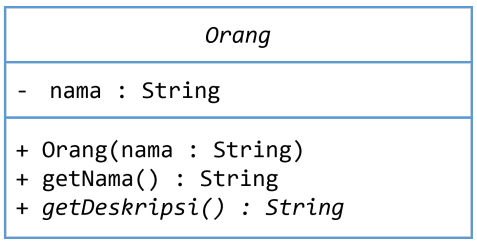
Kemudian, pada baris 20 sampai dengan baris 24, kita menuliskan loop `for` enhanced untuk mengiterasi elemen-elemen array `grup`. Di dalam loop `for` enhanced ini kita menggunakan variabel `o` bertipe data `Orang` sebagai variabel sementara yang menyimpan referensi ke object-object dalam array `grup` saat iterasi. Di dalam loop `for` kita memanggil method `getNama` dan method `getDeskripsi`. Karena kedua method ini didefinisikan di dalam class `Orang` kita dapat memanggil method-method ini melalui variabel `o`. Pemanggilan method `getNama` akan memanggil method `getNama` versi superclass. Sedangkan pemanggilan method `getDeskripsi` akan memanggil method `getDeskripsi` versi subclass karena di dalam subclass-subclass method `getDeskripsi` di-overriding.

Hal-hal penting yang perlu diingat mengenai method abstract dan class abstract:

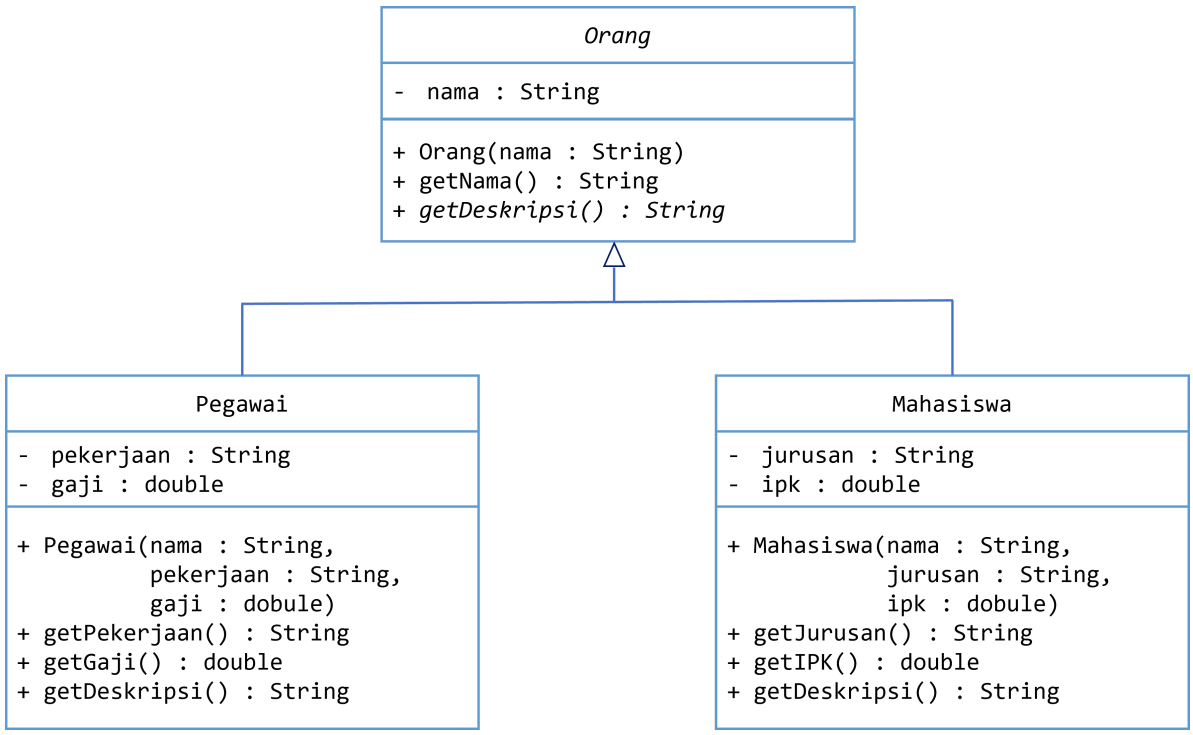
- Method abstract dan class abstract didefinisikan dengan keyword `abstract`.
- Method abstract tidak mempunyai body, dan headernya harus diakhiri dengan titik koma.
- Method abstract harus di-overriding dalam sebuah subclass.
- Jika sebuah class mengandung method abstract, class tersebut tidak dapat di-instansiasi. Class ini berfungsi sebagai superclass.
- Class abstract tidak dapat diinstansiasi. Class abtract berfungsi sebagai superclass.

Menggambarkan Class Abstract dalam Diagram UML

Class abstract digambarkan seperti class-class lainnya dalam UML, hanya saja nama dari class dituliskan miring. Sebagai contoh, gambar berikut adalah diagram UML untuk class abstract `Orang`:



Gambar berikut adalah diagram UML dari class abstract `Orang` beserta subclass-subclassnya:



9.8 Interface

Interface dalam bentuk dasarnya mirip dengan class abstract yang hanya terdiri dari method-method abstract. Di dalam class abstract kita dapat menuliskan method abstract maupun method non-abstract. Namun di dalam interface, kita hanya bisa menuliskan method-method abstract. Method non-abstract tidak diperbolehkan ditulis di dalam interface.

Interface tidak dapat diinstansiasi. Interface harus diimplementasikan oleh class. Class yang mengimplementasikan sebuah interface harus meng-overriding method-method yang dirinci oleh interface tersebut.

Penulisan definisi interface mirip dengan definisi class, hanya saja kita tidak menggunakan keyword `class` tetapi menggunakan keyword `interface`. Method-method yang ditetapkan oleh interface tidak mempunyai body, hanya header yang diakhiri dengan titik koma. Berikut adalah syntax umum dari definisi interface:

```
public interface NamaInterface
{
    Header-header Method...
}
```

Berikut adalah contoh sebuah interface bernama `Bentuk`:

Definisi Interface `Bentuk` (Bentuk.java)

```
public interface Bentuk
{
    double getLuas();
    double getKeliling();
}
```

Interface `Bentuk` merinci dua method: `getLuas` dan `getKeliling`. Perhatikan pada header dari kedua method tersebut, kita tidak menuliskan access specifier. Ini karena semua method dalam interface secara otomatis adalah `public`. Kita dapat saja menuliskan `public` pada header method tetapi praktik yang umum adalah dengan tidak menuliskannya karena semua method dalam interface haruslah public.

Interface mirip dengan class dengan perbedaan-perbedaan berikut:

- Semua method dalam interface haruslah berupa method abstract.
- Semua method dalam interface secara otomatis `public`.
- Interface tidak dapat mempunyai field instance.
- Interface tidak dapat mempunyai method static.

Interface diimplementasi oleh class-class. Kita dapat membayangkan interface seperti daftar method-method yang harus dimiliki oleh class-class yang mengimplementasi interface tersebut. Sebagai contoh, class `Lingkaran` berikut mengimplementasikan interface `Bentuk`:

Definisi Class `Lingkaran` (Lingkaran.java)

```
/*
    Class Lingkaran merepresentasikan bentuk lingkaran.
    Class ini mengimplementasikan interface Bentuk.
*/
public class Lingkaran implements Bentuk
{
    private double radius;

    /*
        Constructor
        @param radius Radius dari lingkaran.
    */
    public Lingkaran(double radius)
    {
        this.radius = radius;
    }

    /*
        Method getRadius mengembalikan radius lingkaran.
        @return Nilai pada field radius.
    */
    public double getRadius()
    {
        return radius;
    }

    /*
        Method getLuas mengembalikan luas lingkaran.
        @return Luas lingkaran.
```

```
    */
    public double getLuas()
    {
        return Math.PI * radius * radius;
    }

    /*
        Method getKeliling mengembalikan keliling lingkaran.
        @return Keliling lingkaran.
    */
    public double getKeliling()
    {
        return 2.0 * Math.PI * radius;
    }
}
```

Untuk mendeklarasikan sebuah class yang mengimplementasikan sebuah interface, kita menuliskan keyword `implements` yang diikuti nama interface yang ingin diimplementasikan setelah nama class pada header class tersebut. Perhatikan pada header dari class di atas, kita menuliskan `implements Bentuk`. Ini berarti class `Lingkaran` mengimplementasikan interface `Bentuk`, sehingga class ini harus menyediakan implementasi (menuliskan body berisi kode-kode) dari method `getLuas` dan method `getKeliling`. Implementasi method `getLuas` kita tuliskan pada baris 11 sampai dengan 14 dan implementasi method `getKeliling` kita tuliskan pada baris 16 sampai dengan 19. Selain method `getLuas` dan `getKeliling`, class `Lingkaran` juga mempunyai method instance `getRadius`, pada baris 18 sampai dengan 21, yang mengembalikan luas lingkaran.

Program berikut mendemonstrasikan class `Lingkaran`:

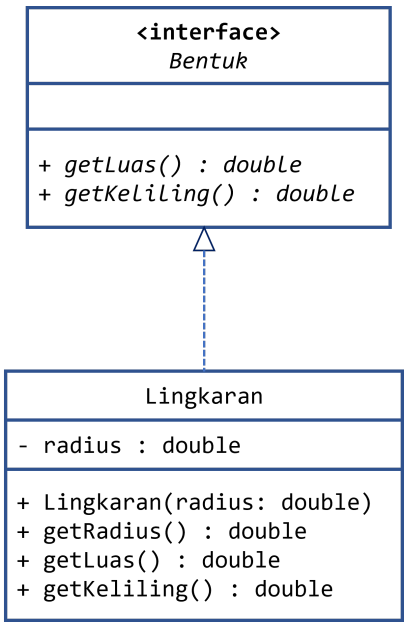
Program (DemoLingkaran.java)

```
/*
    Program ini mendemonstrasikan class Lingkaran.
*/
public class DemoLingkaran
{
    public static void main(String[] args)
    {
        // Buat instance dari class Lingkaran.
        Lingkaran a = new Lingkaran(12);

        // Panggil method getRadius
        System.out.println("Lingkaran dengan radius " + a.getRadius() + ":");

        // Panggil method getLuas
        System.out.println("Luas = " + a.getLuas());

        // Panggil method getKeliling
        System.out.println("Keliling = " + a.getKeliling());
    }
}
```

Output Program (DemoLingkaran.java)

```
Lingkaran dengan radius 12.0:
Luas = 452.3893421169302
Keliling = 75.39822368615503
```

Pada diagram UML, interface digambarkan seperti sebuah class, hanya saja nama interface dan nama method dituliskan miring, dan tag <<>> dtuliskan di atas nama interface. Relasi antara class dan interface disebut sebagai relasi realisasi. Kita menggambarkan relasi realisasi pada diagram UML dengan menghubungkan class dan interface dengan garis putus-putus dengan ujung panah mengarah ke interface. Berikut adalah diagram UML dari interface `Bentuk` dan class `Lingkaran` berserta relasi keduanya:



Polymorphism dan Interface

Java memungkinkan kita membuat variabel bertipe data berupa interface. Variabel bertipe data interface ini dapat digunakan untuk mereferensikan object-object yang mengimplementasi interface tersebut. Ini adalah polymorphism melalui interface. Sebagai contoh, misalkan selain class `Lingkaran`, kita mempunyai class `PersegiPanjang` dan class `Segitiga` yang mengimplementasi interface `Bentuk`. Berikut adalah definisi class `PersegiPanjang`:

```java
/*
    Class PersegiPanjang merepresentasikan bentuk persegi panjang.
    Class ini mengimplementasi interface Bentuk.
*/
public class PersegiPanjang implements Bentuk
{
    private double panjang;
    private double lebar;
    /*
        Constructor
        @param panjang Panjang persegi panjang.
        @param lebar Lebar persegi panjang.
    */
    public PersegiPanjang(double panjang, double lebar)
    {
        this.panjang = panjang;
        this.lebar = lebar;
    }

    /*
        Method getPanjang mengembalikan panjang persegi panjang.
        @return Nilai pada field panjang.
    */
    public double getPanjang()
    {
        return panjang;
    }

    /*
        Method getLebar mengembalikan lebar persegi panjang.
        @return Nilai pada field lebar.
    */
    public double getLebar()
    {
        return lebar;
    }

    /*
        Method getLuas mengembalikan luas persegi panjang.
        @return Luas persegi panjang.
    */
    public double getLuas()
    {
        return panjang * lebar;
    }

    /*
        Method getKeliling mengembalikan keliling persegi panjang.
        @return Keliling persegi panjang.
    */
    public double getKeliling()
    {
        return 2.0 * (panjang + lebar);
    }
}
```

Berikut adalah definisi class `Segitiga`:

```
/*
    Class Segitiga merepresentasikan bentuk segitiga.
    Class ini mengimplementasikan interface Bentuk.
*/
public class Segitiga implements Bentuk
{
    private double a;
    private double b;
    private double c;

    /*
        Constructor
        @param a Sisi 1.
        @param b Sisi 2.
        @param c Sisi 3.
    */
    public Segitiga(double a, double b, double c)
    {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    /*
        Method getLuas mengembalikan luas segitiga.
        @return Luas segitiga.
    */
    public double getLuas()
    {
        double s = (a + b + c) / 2.0;
        return Math.sqrt(s * (s - a) * (s - b) * (s - c));
    }

    /*
        Method getKeliling mengembalikan keliling segitiga.
        @return Keliling segitiga.
    */
    public double getKeliling()
    {
        return a + b + c;
    }
}
```

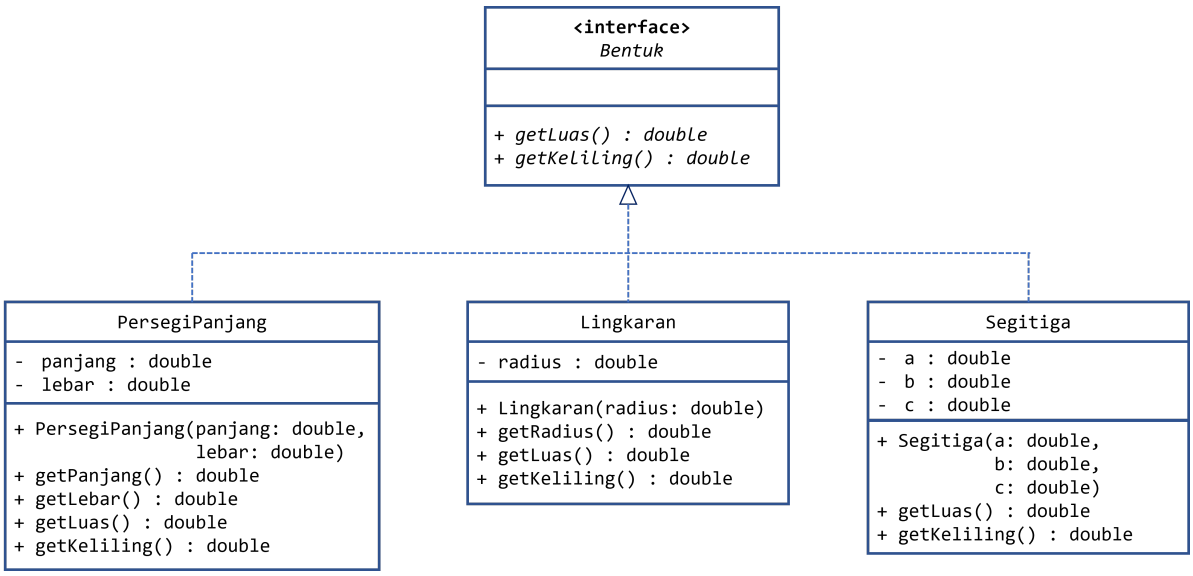
Rumus yang digunakan untuk menghitung luas segitiga pada method `getLuas` adalah rumus Heron yang menyebutkan bahwa luas dari segitiga dengan panjang sisi a, b, dan c dapat dihitung dengan:

$$Luas = \sqrt{s(s-a)(s-b)(s-c)}$$

dimana s adalah setengah keliling segitiga, atau

$$s = \frac{a+b+c}{2}$$

Diagram UML dari interface `Bentuk` dan class-class yang mengimplementasikannya ditunjukkan oleh gambar berikut:



Keuntungan dari interface adalah kita dapat menggunakan polymorphism melalui interface. Kita dapat mempunyai sebuah array bertipe data `Bentuk` dan menugaskan elemen-elemen pada array tersebut untuk mereferensikan class-class yang mengimplementasi interface `Bentuk`. Sebagai contoh, program berikut mendemonstrasikan interface `Bentuk` dan class-class `Lingkaran`, `PersegiPanjang`, dan `Segitiga`.

Program (DemoBentuk.java)

```
/*
    Program ini mendemonstrasikan interface Bentuk.
*/
public class DemoBentuk
{
    public static void main(String[] args)
    {
        Bentuk[] bangun = new Bentuk[3];
        bangun[0] = new PersegiPanjang(18, 18);
        bangun[1] = new Segitiga(30, 30, 30);
        bangun[2] = new Lingkaran(12);

        for (int i = 0; i < bangun.length; i++)
        {
            System.out.println("Luas = " + bangun[i].getLuas() +
                               ", Keliling = " +
                               bangun[i].getKeliling());
        }
    }
}
```

Output Program (DemoBentuk.java)

```
Luas = 324.0, Keliling = 72.0
Luas = 389.7114317029974, Keliling = 90.0
Luas = 452.3893421169302, Keliling = 75.39822368615503
```

Namun, sama seperti polymorphism dengan inheritance, polymorphism dengan interface juga memiliki limitasi: kita hanya dapat memanggil method-method yang dirinci pada interface. Jika sebuah class yang mengimplementasikan interface mempunyai method lain selain yang dirinci oleh interface tersebut, method lain ini tidak dapat dipanggil pada variabel bertipe data interface yang diimplementasikan class tersebut. Misalkan, class `PersegiPanjang` mempunyai method `getPanjang` dan `getLebar`, kedua method ini tidak dirinci pada interface `Bentuk`, sehingga kedua method ini tidak dapat dipanggil melalui elemen array bertipe data `Bentuk`.

Mengimplementasikan Lebih dari Satu Interface

Interface dan class abstract sangatlah mirip, kenapa kita membutuhkan interface? Alasan penggunaan interface adalah sebuah class hanya dapat mengekstensi sebuah class abstract tetapi sebuah class dapat mengimplementasi lebih dari satu interface.

Untuk mendefinisikan class yang mengimplementasi lebih dari satu interface kita menuliskan nama dari interface-interface dipisahkan koma setelah keyword `implements`. Berikut adalah contoh sebuah class yang mengimplementasi lebih dari satu interface:

```
public class MyClass implements Interface1,
                                Interface2,
                                Interface3
{
    // Member-member MyClass
}
```

Field pada Interface

Interface dapat berisi deklarasi-deklarasi field, tetapi semua field-field dalam interface diperlukan sebagai `final` dan `static`. Karena field-field ini secara otomatis menjadi `final` maka kita harus memberikan nilai inisialisasi. Berikut adalah contoh sebuah interface dengan deklarasi field-field:

```
public interface Double
{
    int FIELD1 = 1;
    int FIELD2 = 2;

    // Header-header method...
}
```

Pada interface di atas, `FIELD1` dan `FIELD2` adalah variabel `final static int`. Semua class yang mengimplementasi interface ini akan mempunyai akses ke kedua variabel tersebut.

REFERENSI

[1] Horstmann, Cay S. 2012. *Big Java: Late Objects, 1st Edition*. United States of America: John Wiley & Sons, Inc.

[2] Gaddis, Tony. 2016. *Starting Out with Java: From Control Structures through Objects (6th Edition)*. Boston: Pearson.