Pertemuan IV
Class

4.1. Pengenalan *Class*, *Object,* dan *Method*

4.1.1. *Class*

Class merupakan konsep pokok di bahasa pemrograman berorientasi obyek, demikian pula di java. *Class* mendefinisikan bentuk dan perilaku obyek. Sembarang konsep/abstraksi yang diimplementasikan di java harus dikapsulkan dalam *class*.

Pemrograman di java tidak mungkin dipisahkan dari *class*. Pada pemrograman sebelumnya, *class* hanya mengkapsulkan metode (berbentuk fungsi) *main()* untuk menunjukkan sintaks dasar bahasa java.

Class adalah struktur dasar dari OOP (*Object Oriented Programming*). *Class* terdiri dari dua tipe anggota yang disebut dengan *field* (attribut/properti) dan *method* (metode). *Field* merupakan tipe data yang didefinisikan oleh *class*, sementara *method* merupakan operasi.

4.1.2. *Object*

Sebuah obyek adalah sebuah *instance* (keturunan) dari *class*. Pada dunia perangkat lunak, sebuah obyek adalah sebuah komponen perangkat lunak yang stukturnya mirip dengan obyek pada dunia nyata. Setiap obyek dibangun dari sekumpulan data (atribut) yang disebut variabel untuk menjabarkan karakteristik khusus dari obyek, dan juga terdiri dari sekumpulan *method* yang menjabarkan tingkah laku dari obyek. Bisa dikatakan bahwa obyek adalah sebuah perangkat lunak yang berisi sekumpulan variabel dan *method* yang berhubungan. Variabel dan *method* dalam obyek Java secara formal diketahui sebagai variabel *instance* dan *method instance*. Hal ini dilakukan untuk membedakan dari variabel *class* dan *method class*.

4.1.3. *Method*

Method adalah bagian-bagian kode yang dapat dipanggil oleh program utama atau dari metode lainnya untuk menjalankan fungsi yang spesifik.

Berikut adalah karakteristik dari metode :

a. Dapat mengembalikan satu nilai atau tidak sama sekali
b. Dapat menerima beberapa parameter yang dibutuhkan atau tidak ada parameter sama sekali. Parameter bisa juga disebut sebagai argumen dari fungsi.
c. Setelah metode selesai dieksekusi, dia akan kembali pada metode yang memanggilnya.

4.2. Membuat *Class*

Sebelum menulis *class* Anda, pertama pertimbangkan dimana Anda akan menggunakan *class* dan bagaimana *class* tersebut akan digunakan. Pertimbangkan pula nama yang tepat dan tuliskan seluruh informasi atau properti yang ingin Anda isi pada *class*. Jangan sampai terlupa untuk menuliskan secara urut *method* yang akan Anda gunakan dalam *class*.

Dalam mendefinisikan *class* secara umum dapat ditulis sebagai berikut :

```
<modifier> class <ClassName> {
    <attributeDeclaration>*
    <constructorDeclaration>*
    <methodDeclaration>*
}
```

dimana :

`<modifier>` adalah sebuah *access modifier*, yang dapat dikombinasikan dengan tipe *modifier* lain.

`<ClassName>` adalah nama class yang akan kita buat.

4.3. Deklarasi *Methods*

Sebelum kita membahas *method* apa yang akan dipakai pada *class*, mari kita perhatikan penulisan *method* secara umum.

Dalam pendeklarasian *method*, kita tuliskan :

```
<modifier> <returnType> <name>(<parameter>*) {
    <statement>*
}
```

dimana,

<modifier> dapat menggunakan beberapa *modifier* yang berbeda

<returnType> dapat berupa seluruh tipe data, termasuk *void*

<name> *identifier* atas *class*

<parameter> terdiri dari <tipe_parameter> <nama_parameter>[,]

4.3.1. Accessor Methods

Untuk mengimplementasikan enkapsulasi, kita tidak menginginkan sembarang *object* dapat mengakses data kapan saja. Untuk itu, kita deklarasikan atribut dari *class* sebagai *private*. Namun, ada kalanya dimana kita menginginkan *object* lain untuk dapat mengakses data *private*. Dalam hal ini kita gunakan *accessor methods.*

Accessor Methods digunakan untuk membaca nilai variabel pada *class*, baik berupa *instance* maupun *static*. Sebuah *accessor method* umumnya dimulai dengan penulisan *get<namaInstanceVariable>*. *Method* ini juga mempunyai sebuah *return value*.

Sebagai contoh, kita ingin menggunakan *accessor method* untuk dapat membaca nama, alamat, nilai bahasa Inggris, Matematika, dan ilmu pasti dari siswa.

Mari kita perhatikan salah satu contoh implementasi *accessor method*.

```
public class StudentRecord
{
    private String name;
    :
    :
    :
    public String getName(){
        return name;
    }
}
```

dimana,

public - Menjelaskan bahwa *method* tersebut dapat diakses dari *object* luar class
String - Tipe data *return value* dari *method* tersebut bertipe String
getName - Nama dari *method*
() - Menjelaskan bahwa *method* tidak memiliki parameter apapun

Pernyataan berikut,

```
return name;
```

dalam program kita menandakan akan ada pengembalian nilai dari nama *instance variable* ke pemanggilan *method*. Perhatikan bahwa *return type* dari *method* harus sama dengan tipe data seperti data pada pernyataan *return*. Anda akan mendapatkan pesan kesalahan

sebagai berikut bila tipe data yang digunakan tidak sama :

```
StudentRecord.java:14: incompatible types
found : int
required: java.lang.String
return age;
^
1 error
```

Contoh lain dari penggunaan *accessor method* adalah getAverage,

```
public class StudentRecord
{
  private String name;
  :
  :
  :
  public double getAverage(){
    double result = 0;
    result = ( mathGrade+englishGrade+scienceGrade )/3;
    return result;
  }
}
```

Method getAverage() menghitung rata – rata dari 3 nilai siswa dan menghasilkan nilai *return value* dengan nama *result.*

4.3.2. Mutator Methods

Bagaimana jika kita menghendaki *object* lain untuk mengubah data? Yang dapat kita lakukan adalah membuat *method* yang dapat memberi atau mengubah nilai variable dalam *class*, baik itu berupa *instance* maupun *static*. *Method* semacam ini disebut dengan *mutator methods*. Sebuah *mutator method* umumnya tertulis set<namaInstanceVariabel>.

Mari kita perhatikan salah satu dari implementasi *mutator method* :

```
public class StudentRecord
{
   private String name;
   :
   :
   public void setName( String temp ){
      name = temp;
   }
}
```

Dimana :

public	- Menjelaskan bahwa *method* ini dapat dipanggil *object* luar class
void	- *Method* ini tidak menghasilkan *return value*
setName	- Nama dari *method*
(String temp)	- Parameter yang akan digunakan pada *method*

Pernyataan berikut :

```
name = temp;
```

mengubah nilai *instance variable* name menjadi sama dengan nilai dari temp.

Perlu diingat bahwa *mutator methods* tidak menghasilkan *return value*. Namun berisi beberapa argumen dari program yang akan digunakan oleh *method*.

4.3.3. Multiple *Return Statements*

Anda dapat mempunyai banyak *return values* pada sebuah *method* selama mereka tidak pada blok program yang sama. Anda juga dapat menggunakan konstanta disamping variabel sebagai *return value*.

Sebagai contoh, perhatikan *method* berikut ini :

```
public String getNumberInWords( int num ){
    String defaultNum = "zero";
    if( num == 1 ){
        return "one"; //mengembalikan sebuah konstanta
    }
    else if( num == 2){
        return "two"; //mengembalikan sebuah konstanta
    }
    return defaultNum; // mengembalikan sebuah variabel
}
```

4.3.4. Static Methods

Kita menggunakan *static method* untuk mengakses *static variable* studentCount.

```
public class StudentRecord
{
    private static int studentCount;
    public static int getStudentCount(){
        return studentCount;
    }
}
```

Dimana :

public	- Menjelaskan bahwa *method* ini dapat diakses dari *object* luar class
static	- *Method* ini adalah *static* dan pemanggilannya menggunakan [namaclass].[namaMethod]. Sebagai contoh : studentRecord.getStudentCount
Int	- Tipe *return* dari *method.* Mengindikasikan *method* tersebut harus mempunyai *return value* berupa integer
getStudentCount()	- Nama dari *method* - *Method* ini tidak memiliki parameter apapun

Pada deklarasi di atas, *method* getStudentCount() akan selalu menghasilkan *return value* 0 jika kita tidak mengubah apapun pada kode program untuk mengatur nilainya. Kita akan membahas pengubahan nilai dari studentCount pada pembahasan *constructor*.

4.4. Referensi *this*

Reference *this* digunakan untuk mengakses instance variable yang dibiaskan oleh parameter. Untuk pemahaman lebih lanjut, mari kita perhatikan contoh pada *method* setAge. Dimisalkan kita mempunyai kode deklarasi berikut pada *method* setAge.

```
public void setAge( int age ){
    age = age; //SALAH!!!
}
```

Nama parameter pada deklarasi ini adalah age, yang memiliki penamaan yang sama dengan *instance variable* age. Parameter age adalah deklarasi terdekat dari *method*, sehingga nilai dari parameter tersebut akan digunakan. Maka pada pernyataan :

```
age = age;
```

kita telah menentukan nilai dari parameter age kepada parameter itu sendiri. Hal ini sangat tidak kita kehendaki pada kode program kita. Untuk menghindari kesalahan semacam ini, kita gunakan metode referensi *this*. Untuk menggunakan tipe referensi ini, kita tuliskan :

```
this.<namaInstanceVariable>
```

Sebagai contoh, kita dapat menulis ulang kode hingga tampak sebagai berikut :

```
public void setAge( int age ){
    this.age = age;
}
```

Method ini akan mereferensikan nilai dari parameter age kepada *instance variable* dari *object* StudentRecord.

Kita hanya dapat menggunakan referensi this terhadap *instance variable* dan bukan *static* ataupun class variabel.

4.5. *Overloading Methods*

Dalam *class* yang kita buat, kadangkala kita menginginkan untuk membuat *method* dengan nama yang sama namun mempunyai fungsi yang berbeda menurut parameter yang digunakan. Kemampuan ini dimungkinkan dalam pemrograman Java, dan dikenal sebagai *overloading method*.

Overloading method mengijinkan sebuah *method* dengan nama yang sama namun memiliki parameter yang berbeda sehingga mempunyai implementasi dan *return value* yang berbeda pula. Daripada memberikan nama yang berbeda pada setiap pembuatan *method*, *overloading method* dapat digunakan pada operasi yang sama namun berbeda dalam implementasinya.

Sebagai contoh, pada *class* StudentRecord kita menginginkan sebuah *method* yang akan menampilkan informasi tentang siswa. Namun kita juga menginginkan operasi penampilan data tersebut menghasilkan *output* yang berbeda menurut parameter yang digunakan. Jika pada saat kita memberikan sebuah parameter berupa string, hasil yang ditampilkan adalah nama, alamat dan umur dari siswa, sedang pada saat kita memberikan 3 nilai dengan tipe *double*, kita menginginkan *method* tersebut untuk menampilkan nama dan nilai dari siswa.

Untuk mendapatkan hasil yang sesuai, kita gunakan *overloading method* di dalam deklarasi *class* StudentRecord.

```
public void print( String temp ){
    System.out.println("Name:" + name);
    System.out.println("Address:" + address);
    System.out.println("Age:" + age);
}

public void print(double eGrade, double mGrade, double sGrade){
    System.out.println("Name:" + name);
    System.out.println("Math Grade:" + mGrade);
    System.out.println("English Grade:" + eGrade);
    System.out.println("Science Grade:" + sGrade);
}
```

Jika kita panggil pada *method* utama *(main)* :

```
public static void main( String[] args )
{
    StudentRecord annaRecord = new StudentRecord();
    annaRecord.setName("Anna");
    annaRecord.setAddress("Philippines");
    annaRecord.setAge(15);
    annaRecord.setMathGrade(80);
    annaRecord.setEnglishGrade(95.5);
    annaRecord.setScienceGrade(100);

    //overloaded methods
    //panggilan metode print pertama
    annaRecord.print( annaRecord.getName() );

    //panggilan metode print kedua
    annaRecord.print( annaRecord.getEnglishGrade(),
    annaRecord.getMathGrade(), annaRecord.getScienceGrade());
}
```

Kita akan mendapatkan *output* pada panggilan metode print pertama sebagai berikut :

```
Name:Anna
Address:Philippines
Age:15
```

Kemudian akan dihasilkan *output* sebagai berikut pada panggilan metode print kedua :

```
Name:Anna
Math Grade:80.0
English Grade:95.5
Science Grade:100.0
```

Jangan lupakan bahwa *overloaded method* memiliki *property* sebagai berikut :

a. Nama yang sama
b. Parameter yang berbeda
c. Nilai kembalian (*return*) bisa sama ataupun berbeda

Contoh program :

```
public class Segitiga{
   private double tinggi;
   private double alas;

   public void settinggi(double tinggi){
      this.tinggi = tinggi;
   }

   public void setalas(double alas){
      this.alas = alas;
   }

   public double gettinggi(){
      return tinggi;
   }

   public double getalas(){
      return alas;
   }

   public double getluas(){
      return (this.tinggi * this.alas * 0.5);
   }

   public static void main (String args[]){
      Segitiga S[] = new Segitiga[2];
      Byte i;

      //Membuat objek dari class Segitiga
      S[0] = new Segitiga();
      S[1] = new Segitiga();

      S[0].settinggi(12.0);
      S[0].setalas(8.0);
```

```
      S[1].settinggi(11.23);
      S[1].setalas(7.7);

      for (i=0;i<2;i++){
        System.out.println("Segitiga ke-" + (i+1));
        System.out.println("Tinggi = " + S[i].gettinggi());
        System.out.println("Alas = " + S[i].getalas());
        System.out.println("Luas Segitiga = " + S[i].getluas());
        System.out.println();
      }
    }
}
```

4.6. Deklarasi *Constructor*

Telah tersirat pada pembahasan sebelumnya, *constructor* sangatlah penting pada pembentukan sebuah *object*. *Constructor* adalah *method* dimana seluruh inisialisasi *object* ditempatkan.

Berikut ini adalah *property* dari *constructor* :

a. *Constructor* memiliki nama yang sama dengan *class*
b. Sebuah *constructor* mirip dengan *method* pada umumnya, namun hanya informasi – informasi berikut yang dapat ditempatkan pada *header* sebuah *constructor*, *scope* atau identifikasi pengaksesan (misal: *public*), nama dari konstuktor dan parameter.
c. *Constructor* tidak memiliki *return value*
d. *Constructor* tidak dapat dipanggil secara langsung, namun harus dipanggil dengan menggunakan operator *new* pada pembentukan sebuah objek.

Untuk mendeklarasikan *constructor*, kita dapat menggunakan struktur penulisan secara umum seperti berikut ini :

```
<modifier> <className> (<parameter>*) {
    <statement>
    <statement>
    :
    :
}
```

4.6.1. Default Constructor

Setiap class memiliki default constructor. Sebuah default constructor adalah constructor yang tidak memiliki parameter apapun. Jika sebuah *class* tidak memiliki constructor apapun, maka sebuah default constructor akan dibentuk secara implisit :

Sebagai contoh, pada *class* StudentRecord, bentuk default constructor akan terlihat seperti dibawah ini :

```
public StudentRecord()
{
    //area penulisan kode
}
```

4.6.2. Overloading Constructor

Seperti telah kita bahas sebelumnya, sebuah *constructor* juga dapat dibentuk menjadi *overloaded*. Dapat dilihat pada 4 contoh sebagai berikut :

```
public StudentRecord(){
    //area inisialisasi kode;
}

public StudentRecord(String temp){
    this.name = temp;
}

public StudentRecord(String name, String address){
    this.name = name;
    this.address = address;
}

public StudentRecord(double mGrade, double eGrade, double sGrade){
    mathGrade = mGrade;
    englishGrade = eGrade;
    scienceGrade = sGrade;
}
```

4.6.3. Menggunakan Constructor

Untuk menggunakan constructor, kita gunakan kode – kode sebagai berikut :

```
public static void main( String[] args )
{
   //membuat 3 objek
   StudentRecord annaRecord=new StudentRecord("Anna");
   StudentRecord beahRecord=new StudentRecord("Beah","Philippines");
   StudentRecord crisRecord=new StudentRecord(80,90,100);

   //area penulisan kode selanjtunya
}
```

Sebelum kita lanjutkan, mari kita perhatikan kembali deklarasi variabel static studentCount yang telah dibuat sebelumnya. Tujuan deklarasi studentCount adalah untuk menghitung jumlah *object* yang dibentuk pada *class* StudentRecord. Jadi, apa yang akan kita lakukan selanjutnya adalah menambahkan nilai dari studentCount setiap kali setiap pembentukan *object* pada *class* StudentRecord. Lokasi yang tepat untuk memodifikasi dan menambahkan nilai studentCount terletak pada constructor-nya, karena selalu dipanggil setiap kali objek terbentuk.

Sebagai contoh :

```
public StudentRecord(){
   //letak kode inisialisasi
   studentCount++; //menambah student
}

public StudentRecord(String temp){
   this.name = temp;
   studentCount++; //menambah student
}

public StudentRecord(String name, String address){
   this.name = name;
   this.address = address;
   studentCount++; //menambah student
}
```

```
public StudentRecord(double mGrade, double eGrade, double sGrade){
   mathGrade = mGrade;
   englishGrade = eGrade;
   scienceGrade = sGrade;
   studentCount++; //menambah student
}
```

4.6.4. Pemanggilan Constructor dengan this()

Pemanggilan constructor dapat dilakukan secara berangkai, dalam arti Anda dapat memanggil constructor di dalam constructor lain. Pemanggilan dapat dilakukan dengan referensi *this()*. Perhatikan contoh kode sebagai berikut :

```
1:  public StudentRecord(){
2:  this("some string");
3:
4:  }
5:
6:  public StudentRecord(String temp){
7:  this.name = temp;
8:  }
9:
10: public static void main( String[] args )
11: {
12:
13: StudentRecord annaRecord = new StudentRecord();
14: }
```

Dari contoh kode diatas, pada saat baris ke 13 dipanggil akan memanggil constructor dasar pada baris pertama. Pada saat baris kedua dijalankan, baris tersebut akan menjalankan constructor yang memiliki parameter String pada baris ke-6.

Beberapa hal yang patut diperhatikan pada penggunaan *this()* :

a. Harus dituliskan pada baris pertama pada sebuah constructor
b. Hanya dapat digunakan pada satu definisi constructor. Kemudian metode ini dapat diikuti dengan kode – kode berikutnya yang relevan

4.7. Package

Packages dalam JAVA berarti pengelompokan beberapa *class* dan *interface* dalam satu unit. Fitur ini menyediakan mekanisme untuk mengatur *class* dan *interface* dalam jumlah banyak dan menghindari konflik pada penamaan.

4.7.1. Mengimport Package

Supaya dapat meggunakan *class* yang berada diluar *package* yang sedang dikerjakan, Anda harus mengimport *package* dimana *class* tersebut berada. Pada dasarnya, seluruh program JAVA mengimport *package java.lang.*,* sehingga Anda dapat menggunakan *class* seperti String dan Integer dalam program meskipun belum mengimport *package* sama sekali.

Penulisan import *package* dapat dilakukan seperti dibawah ini :

```
import <namaPaket>;
```

Sebagai contoh, bila Anda ingin menggunakan *class* Color dalam *package* awt, Anda harus menuliskan import *package* sebagai berikut :

```
import java.awt.Color;
import java.awt.*;
```

Baris pertama menyatakan untuk mengimport *class* Color secara spesifik pada *package*, sedangkan baris kedua menyatakan mengimport seluruh *class* yang terkandung dalam *package java.awt*.

Cara lain dalam mengimport *package* adalah dengan menuliskan referensi *package* secara eksplisit. Hal ini dilakukan dengan menggunakan nama *package* untuk mendeklarasikan *object* sebuah *class* :

```
java.awt.Color color;
```

4.7.2. Membuat Package

Untuk membuat *package*, dapat dilakukan dengan menuliskan :

```
package <packageName>;
```

Anggaplah kita ingin membuat *package* dimana *class* StudentRecord akan ditempatkan bersama dengan *class* – *class* yang lain dengan nama *package* schoolClasses.

Langkah pertama yang harus dilakukan adalah membuat folder dengan nama schoolClasses. Salin seluruh *class* yang ingin diletakkan pada *package* dalam folder ini.

Kemudian tambahkan kode deklarasi *package* pada awal file. Sebagai contoh :

```
package schoolClasses;
public class StudentRecord
{
    private String name;
    private String address;
    private int age;
}
```

Package juga dapat dibuat secara bersarang. Dalam hal ini Java Interpreter menghendaki struktur direktori yang mengandung *class* eksekusi untuk disesuaikan dengan struktur *package*.

4.7.3. Pengaturan CLASSPATH

Diasumsikan *package* schoolClasses terdapat pada direktori C:\. Langkah selanjutnya adalah mengatur classpath untuk menunjuk direktori tersebut sehingga pada saat akan dijalankan, JVM dapat mengetahui dimana *class* tersebut tersimpan.

Sebelum membahas cara mengatur classpath, perhatikan contoh dibawah yang menandakan kejadian bila kita tidak mengatur classpath.

Asumsikan kita mengkompilasi dan menjalankan *class* StudentRecord :

```
C:\schoolClasses>javac StudentRecord.java
C:\schoolClasses>java StudentRecord
Exception in thread "main" java.lang.NoClassDefFoundError: StudentRecord
(wrong name: schoolClasses/StudentRecord)
at java.lang.ClassLoader.defineClass1(Native Method)
at java.lang.ClassLoader.defineClass(Unknown Source)
at java.security.SecureClassLoader.defineClass(Unknown Source)
at java.net.URLClassLoader.defineClass(Unknown Source)
at java.net.URLClassLoader.access$100(Unknown Source)
at java.net.URLClassLoader$1.run(Unknown Source)
```

```
at java.security.AccessController.doPrivileged(Native Method)
at java.net.URLClassLoader.findClass(Unknown Source)
at java.lang.ClassLoader.loadClass(Unknown Source)
at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
at java.lang.ClassLoader.loadClass(Unknown Source)
at java.lang.ClassLoader.loadClassInternal(Unknown Source)
```

Kita akan mendapatkan pesan kesalahan berupa NoClassDefFoundError yang berarti JAVA tidak mengetahui dimana posisi *class*. Hal tersebut disebabkan oleh karena *class* StudentRecord berada pada *package* dengan nama studentClasses. Jika kita ingin menjalankan class tersebut, kita harus memberi informasi pada JAVA bahwa nama lengkap dari *class* tersebut adalah schoolClasses.StudentRecord. Kita juga harus menginformasikan kepada JVM dimana posisi pencarian *package*, yang dalam hal ini berada pada direktori C:\. Untuk melakukan langkah – langkah tersebut, kita harus mengatur *classpath*.

Pengaturan classpath pada Windows dilakukan pada *command prompt* :

```
C:\schoolClasses> set classpath=C:\
```

dimana C:\ adalah direktori dimana kita menempatkan *package* (Buatlah *folder*/direktori sesuai nama *package* didalam *classpath*). Setelah mengatur *classpath*, kita dapat menjalankan program di mana saja dengan mengetikkan :

```
C:\schoolClasses> java schoolClasses.StudentRecord
```

Pada UNIX, asumsikan bahwa kita memiliki *class* - *class* yang terdapat dalam direktori /usr/local/myClasses, ketikkan :

```
export classpath=/usr/local/myClasses
```

Perhatikan bahwa Anda dapat mengatur *classpath* dimana saja. Anda juga dapat mengatur lebih dari satu *classpath*, kita hanya perlu memisahkannya dengan menggunakan **;** (Windows), dan **:** (UNIX). Sebagai contoh :

```
set classpath=C:\myClasses;D:\;E:\MyPrograms\Java
```

dan untuk sistem UNIX :

```
export classpath=/usr/local/java:/usr/myClasses
```

4.8. *Access Modifiers*

Pada saat membuat, mengatur *properties* dan *class methods*, kita ingin untuk mengimplementasikan beberapa macam larangan untuk mengakses data. Sebagai contoh, jika Anda ingin beberapa atribut hanya dapat diubah hanya dengan *method* tertentu, tentu Anda ingin menyembunyikannya dari *object* lain pada *class*. Di JAVA, implementasi tersebut disebut dengan *access modifiers.*

Terdapat 4 macam *access modifiers* di JAVA, yaitu : *public, private, protected* dan *default*. 3 tipe akses pertama tertulis secara ekplisit pada kode untuk mengindikasikan tipe akses, sedangkan yang keempat yang merupakan tipe *default*, tidak diperlukan penulisan *keyword* atas tipe.

4.8.1. Akses *Default*

Tipe ini mensyaratkan bahwa hanya *class* dalam *package* yang sama yang memiliki hak akses terhadap variabel dan *methods* dalam *class*. Tidak terdapat *keyword* pada tipe ini.

Sebagai contoh :

```
public class StudentRecord
{
    //akses dasar terhadap variabel
    int name;
    //akses dasar terhadap metode
    String getName(){
        return name;
    }
}
```

Pada contoh diatas, variabel nama dan *method* getName() dapat diakses dari *object* lain selama *object* tersebut berada pada *package* yang sama dengan letak dari file StudentRecord.

4.8.2. Akses Public

Tipe ini mengijinkan seluruh *class member* untuk diakses baik dari dalam dan luar *class*. *Object* apapun yang memiliki interaksi pada *class* memiliki akses penuh terhadap *member* dari tipe ini.

Sebagai contoh :

```
public class StudentRecord
{
    //akses dasar terhadap variabel
    public int name;
    //akses dasar terhadap metode
    public String getName(){
        return name;
    }
}
```

Dalam contoh ini, variabel name dan *method* getName() dapat diakses dari *object* lain.

4.8.3. Akses Protected

Tipe ini hanya mengijinkan *class member* untuk diakses oleh *method* dalam *class* tersebut dan elemen – elemen *subclass*. Sebagai contoh :

```
public class StudentRecord
{
    //akses pada variabel
    protected int name;
    //akses pada metode
    protected String getName(){
        return name;
    }
}
```

Pada contoh diatas, variabel name dan *method* getName() hanya dapat diakses oleh *method* internal *class* dan *subclass* dari *class* StudentRecord. Definisi *subclass* akan dibahas pada bab selanjutnya.

4.8.4. Akses *Private*

Tipe ini mengijinkan pengaksesan *class* hanya dapat diakses oleh *class* dimana tipe ini dibuat. Sebagai contoh :

```
public class StudentRecord
{
    //akses dasar terhadap variabel
    private int name;
    //akses dasar terhadap metode
    private String getName(){
        return name;
    }
}
```

Pada contoh diatas, variabel name dan *method* getName() hanya dapat diakses oleh *method* internal *class* tersebut.

Instance variable dari *class* secara *default* akan bertipe *private* sehingga *class* tersebut hanya akan menyediakan *accessor* dan *mutator methods* terhadap variabel ini.

Contoh program :

```
public class Segitiga{
  private double tinggi;
  private double alas;


  public Segitiga(){
    tinggi=0;
     alas=0;
  }

  /*public Segitiga(double initTinggi, double initAlas){
    tinggi=initTinggi;
     alas=initAlas;
  }*/

  public Segitiga(double tinggi, double alas){
    this.tinggi=tinggi;
     this.alas=alas;
  }

  public void settinggi(double tinggi){
    this.tinggi = tinggi;
  }

  public void setalas(double alas){
    this.alas = alas;
  }

  public double gettinggi(){
    return tinggi;
  }
```

```
  public double getalas(){
    return alas;
  }

  public double getluas(){
    return (this.tinggi * this.alas * 0.5);
  }

  public static void main (String args[]){
    Segitiga S[] = new Segitiga[4];
     Byte i;
     String s="";

     S[0] = new Segitiga(6,8);
     S[1] = new Segitiga(5,3);
     S[2] = new Segitiga();
     S[3] = new Segitiga();

     S[2].settinggi(12.0);
     S[2].setalas(8.0);

     S[3].settinggi(11.23);
     S[3].setalas(7.7);

     for (i=0;i<4;i++){
       System.out.println("Segitiga ke-" + (i+1));
       System.out.println("Tinggi = " + S[i].gettinggi());
       System.out.println("Alas = " + S[i].getalas());
       System.out.println("Luas Segitiga = " + S[i].getluas());
       System.out.println();
     }
  }
}
```

Contoh program yang memanggil Class program sebelumnya :

```
public class HitungSegitiga{

  public static void main (String args[]){
     Segitiga S1 = new Segitiga();
     Segitiga S2 = new Segitiga(10,5);

     S1.settinggi(12.0);
     S1.setalas(9.0);

     System.out.println("Tinggi = " + S1.gettinggi());
     System.out.println("Alas = " + S1.getalas());
     System.out.println("Luas Segitiga = " + S1.getluas());
     System.out.println();
     System.out.println("Tinggi = " + S2.gettinggi());
     System.out.println("Alas = " + S2.getalas());
     System.out.println("Luas Segitiga = " + S2.getluas());
  }
}
```

Referensi:

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