MODUL - OBJECT ORIENTED PROGRAMMING 1

Oleh :

Priyanto Tamami, S.Kom.

POLITEKNIK HARAPAN BERSAMA

PROGRAM STUDI D IV TEKNIK INFORMATIKA

Jl. Mataram No. 9 Tegal

Daftar Isi

Bab 1

Kelas dan Objek

1.1 Tujuan

Pada Bab ini diharapkan mahasiswa memahami pengertian dan perbedaan Kelas dan Objek dan mampu mengimplementasikan konsep tersebut pada bahasa pemrograman Java.

1.2 Pengantar

Dalam paradigma pemrograman berorientasi objek, untuk membangun sebuah sistem atau aplikasi yang lengkap, sistem tersebut akan dipecah menjadi bagian-bagian kecil yang disebut dengan objek. Tiap-tiap objek yang terbentuk akan dapat saling berinteraksi membentuk sebuah sistem yang dapat digunakan.

Untuk mempermudah pembentukan objek-objek yang akan digunakan, maka diperlukan klasifikasi-klasifikasi tertentu berdasarkan kesamaan ciri dan fitur, yang disebut dengan kelas. Dengan kata lain bahwa kelas itu sebetulnya adalah deklarasi dari beberapa objek yang nantinya akan digunakan dalam membangun sebuah sistem.

Bagaimana implementasi kedua istilah tersebut dalam bahasa pemrograman Java, mari kita lanjutkan ke bagian **Praktek**.

1.3 Praktek

1.3.1 Kelas

Apabila kita menggunakan bahasa pemrograman Java, aturan yang harus kita ikuti adalah pada saat pembentukan deklarasi sebuah kelas, nama kelas dan nama berkas yang dibuat harus sama persis sampai ke besar dan kecilnya huruf.

Sebagai contoh, apabila kita ingin membuat sebuah kelas `Mahasiswa`, maka kita akan membuat sebuah berkas dengan nama `Mahasiswa.java`, yang didalamnya akan berisi kode berikut :

```
1 public class Mahasiswa {}
```

Agar sebuah aplikasi dapat dijalankan dan dilihat hasilnya, maka kita perlu menambahkan sebuah *method* dengan nama `main` di dalam kelas tersebut, sehingga isi kodenya akan menjadi seperti berikut :

```
1 public class Mahasiswa{
2   public static void main(String args[]) {
3   }
4 }
```

Seluruh aksi program yang dijalankan akan dimulai dari *method* `main` ini. Misalkan kita coba agar aplikasi dapat menampilkan tulisan selamat datang apabila dijalankan, kodenya akan kita ubah menjadi seperti berikut :

```
2 public class Mahasiswa {
3   public static void main(String args[]) {
4     System.out.println("Selamat datang pada mata kuliah OOP");
5   }
```

```
}
```

Agar kode tersebut dapat berjalan, maka kita harus melakukan *compile* terlebih dahulu pada kode sumber dengan cara berikut :

```
$ javac Mahasiswa.java
```

Dari hasil *compile* tersebut, akan terbentuk sebuah berkas dengan nama yang sama, yaitu `Mahasiswa` namun dengan ekstensi `.class`, bila sudah tersebut berkas ini, kita dapat menjalankannya dengan perintah berikut :

```
$ java Mahasiswa
```

Hasil keluaran dari perintah tersebut seharusnya akan menampilkan teks seperti ini :

```
Selamat datang pada mata kuliah OOP
```

1.3.2 Objek

Dari contoh kode sebelumnya, deklarasi kelas `Mahasiswa` sudah memiliki sebuah fitur atau *method* dengan nama `main`. Sekarang kita akan coba menambahkan ciri atau atribut lain pada kelas `Mahasiswa`.

Seorang Mahasiswa tentunya akan memiliki **nama** dan **NIM** (Nomor Induk Mahasiswa), untuk mengimplementasikan atribut ini, kita akan ubah kodenya menjadi seperti ini :

```
public class Mahasiswa {

  String nama;
  String nim;

}
```

Kita hapus terlebih dahulu *method* `main`, agar kita fokus pada kelas `Mahasiswa`. Kelas ini memiliki atribut `nama` dan `nim`, pada kelas ini akan kita tambahkan sebuah fitur atau *method* untuk menampilkan informasi dari Mahasiswa yang bersangkutan, kodenya akan kita tambahkan sehingga terlihat seperti berikut :

```
1  public class Mahasiswa {
2
3    String nama;
4    String nim;
5
6    public void cetakInfo() {
7      System.out.println("Nama : " + nama);
8      System.out.println("NIM : " + nim);
9    }
10
11 }
```

Kelas `Mahasiswa` kita anggap sudah lengkap untuk sementara, kita akan coba membuat sebuah objek dari kelas `Mahasiswa` ini. Buatlah sebuah kelas baru, kita beri nama untuk berkasnya adalah `Aplikasi.java` yang isinya seperti berikut :

```
1  public class Aplikasi {
2    public static void main(String args[]) {
3      Mahasiswa ami = new Mahasiswa();
4      ami.nama = "tamami";
5      ami.nim = "19001";
6      ami.cetakInfo();
7    }
8  }
```

Perhatikan pada baris ke-3, bahwa objek `ami` telah kita buat dengan tipe data berupa kelas `Mahasiswa`, ini artinya, objek `ami` merupakan instan dari kelas `Mahasiswa`.

Pembentukan objek, agar data di dalamnya dapat kita ubah, kita perlu me-

lakukan inisiasi dengan pemanggilan konstruktor `Mahasiswa` dengan kode `new Mahasiswa()`. Konstruktor ini akan kita bahas di bagian lain, namun secara *default*, setiap kelas pasti memiliki 1 (satu) konstruktor tanpa parameter walau tidak dideklarasikan secara eksplisit.

Baris ke-4 dan baris ke-5 mengisikan nilai ke properti `nama` dan `nim` milik objek `ami`. Kemudian pada baris ke-6, *method* `cetakInfo()` milik objek `ami` dipanggil.

Untuk melakukan kompilasi, seperti langkah sebelumnya, kita dapat melakukannya dengan perintah `javac` dari konsol atau *command prompt* seperti berikut :

```
1 $ javac Aplikasi.java
```

Kemudian jalankan dengan perintah berikut :

```
1 $ java Aplikasi
```

Hasil yang dikeluarkan seharusnya akan terlihat seperti berikut :

```
1 Nama : Tamami
2 NIM : 19001
```

1.4 Kesimpulan

Bahwa kelas dan objek itu adalah dua hal yang berbeda, dimana kelas adalah deklarasi sebuah unit yang memiliki atribut dan fitur tertentu, sementara objek adalah instan dari suatu kelas.

1.5 Tugas

Buatlah sebuah kelas `Anggota`, yang di dalamnya terdapat atribut **nomor anggota** dan **nama**. Kemudian buat sebuah objek yang merupakan instan dari kelas

`Anggota` dan isikan *nama* dan *nomor anggota*nya. Kemudian cetak hasilnya dalam format `no. anggota : nama` seperti contoh berikut :

```
19001 : tamami
```

Bab 2

Konstruktor, Field, dan Overloading

2.1 Tujuan

Pada Bab ini diharapkan mahasiswa memahami konsep Konstruktor, *Field*, dan *Overloading* pada bahasa pemrograman Java.

2.2 Pengantar

Pada Bab sebelumnya, kita sempat menyinggung sedikit tentang konstruktor, bahwa setiap kelas yang kita deklarasikan, secara implisit akan menyediakan sebuah konstruktor tanpa parameter di dalamnya, konstruktor ini dipanggil pada saat akan membuat instan bagi sebuah objek.

Namun demikian, konstruktor ini pun sebetulnya dapat kita deklarasikan yang biasanya digunakan untuk memberikan nilai-nilai *default* bagi atribut / *field* yang ada di dalamnya.

Lalu apa itu *field*? *Field* atau atribut sebetulnya sudah sangat kita kenal dalam

konsep paradigma pemrograman yang lain dengan nama *variabel*. Biasanya pada sebuah kelas akan memiliki 1 (satu) atau lebih *field* atau atribut, bersama dengan *method* akan menjadi ciri sebuah kelas.

Selain deklarasi konstruktor tanpa parameter, sebetulnya kita masih dapat mendeklarasikan konstruktor lain dengan parameter, dan dapat dideklarasikan lebih dari 1 (satu) konstruktor, implementasi ini disebut *overloading*.

Mari kita lihat implementasi dari ketiga istilah di atas dalam bahasa pemrograman Java.

2.3 Praktek

2.3.1 Konstruktor

Konstruktor sebetulnya adalah fungsi atau *method* yang dipanggil ketika akan membuat sebuah instan dari kelas. Ciri yang terlihat pada konstruktor ini dibanding *method* lain adalah namanya akan sama persis dengan nama kelasnya, dan tidak memiliki nilai balik sama sekali.

Mari kita lihat contoh kelas *Mahasiswa* sebelumnya seperti berikut :

```
1  public class Mahasiswa {
2
3    String nama;
4    String nim;
5
6    public Mahasiswa() {}
7
8    public void cetakInfo() {
9      System.out.println("Nama : " + nama);
10     System.out.println("NIM : " + nim);
11   }
12
```

```
13 }
```

Tampak pada kode tersebut, pada baris ke-6 adalah deklarasi konstruktor tanpa parameter yang apabila tidak dideklarasikan pun, konstruktor tersebut secara implisit sudah ada. Namun sekarang kita akan modifikasi konstruktor tersebut untuk mengisikan nilai *default* ke atribut `nama` dan `nim`. Kodenya akan menjadi seperti berikut :

```
1  public class Mahasiswa {
2
3    String nama;
4    String nim;
5
6    public Mahasiswa() {
7      nama = "tidak ada";
8      nim = "00000";
9    }
10
11   public void cetakInfo() {
12     System.out.println("Nama : " + nama);
13     System.out.println("NIM : " + nim);
14   }
15
16 }
```

Kita akan melihat perubahan pada baris ke-6 sampai ke-9, konstruktor yang tadinya kosong, tanpa deklarasi isi sama sekali, sekarang kita memberikan nilai *default* pada atribut `nama` dan `nim`.

Sekarang kita coba modifikasi kelas `Aplikasi` dari Bab sebelumnya untuk melihat hasilnya, bagaimana bila instan yang terbentuk tidak kita isikan atribut-atributnya. Berikut kodenya :

```
1  public class Aplikasi {
```

```
2   public static void main(String args[]) {
3     Mahasiswa ami = new Mahasiswa();
4     ami.cetakInfo();
5   }
6 }
```

Perhatikan pada baris ke-3 dan ke-4, objek `ami` hanya membentuk instan baru dengan memanggil konstruktor `Mahasiswa` tanpa parameter, kemudian *method* `cetakInfo()` langsung dipanggil.

Untuk melihat hasil keluarannya, pastikan untuk melakukan *compile* terhadap kelas `Mahasiswa` dan `Aplikasi`. Hasil yang didapat pada layar monitor seharusnya akan terlihat seperti berikut :

```
1 Nama : tidak ada
2 NIM : 00000
```

Hal ini disebabkan karena pada saat kita membentuk instan baru dengan memanggil konstruktor `Mahasiswa` tanpa parameter, nilai atribut `nama` dan `nim` sudah terisi secara otomatis dengan nilai *default*, sehingga apabila tidak ada perubahan, maka hasil yang ditampilkan adalah hasil dari pengisian nilai *default* pada konstruktornya.

2.3.2 Field

Seperti dijelaskan pada bagian sebelumnya bahwa istilah *field* ini lebih kita kenal dengan istilah *variabel* atau dalam istilah yang sering disebut dalam beberapa sumber adalah atribut.

Sehingga pada kelas `Mahasiswa`, atribut atau *field* yang dimiliki adalah `nama` dan `nim`.

Namun hendaknya, sesuai aturan pada desain orientasi objek bahwa akses terhadap *field* ini seharusnya terbatas hanya pada kelas yang bersangkutan, apabila

objek lain ingin melakukan akses atau manipulasi data, maka harus dilakukan melalui *method* yang dapat diakses oleh publik.

Jadi idealnya, bentuk kode dari kelas `Mahasiswa` akan menjadi seperti berikut :

```
1  public class Mahasiswa {
2
3    private String nama;
4    private String nim;
5
6    public Mahasiswa() {
7      nama = "tidak ada";
8      nim = "00000";
9    }
10
11   public void cetakInfo() {
12     System.out.println("Nama : " + nama);
13     System.out.println("NIM : " + nim);
14   }
15
16   public void setNama(String nama) {
17     this.nama = nama;
18   }
19
20   public String getNama() {
21     return nama;
22   }
23
24   public void setNim(String nim) {
25     this.nim = nim;
26   }
27
28   public String getNim() {
```

```
29      return nim;
30    }
31
32  }
```

Terlihat sedikit lebih panjang, namun dari baris ke-16 sampai ke bawah sebetulnya adalah deklarasi aksesor untuk *field* atau atribut `nama` dan `nim` yang menjadi `private` pada baris ke-3 dan ke-4.

Dengan kondisi demikian, diharapkan nilai atribut yang berada di dalam kelas lebih dapat dikontrol, dan pengguna berikutnya tidak perlu terlalu pusing memikirkan detail kelasnya, cukup fokus dengan apa fungsinya.

Dengan kondisi ini pula, maka akses terhadap atribut `nama` dan `nim` dari kelas `Aplikasi` tidak bisa dilakukan dengan cara yang lama, perubahan pada kelas `Aplikasi` menjadi seperti berikut :

```
1  public class Aplikasi {
2    public static void main(String args[]) {
3      Mahasiswa ami = new Mahasiswa();
4      ami.setNama("Tamami");
5      ami.setNim("19001");
6      ami.cetakInfo();
7    }
8  }
```

Perhatikan cara akses terhadap atribut `nama` dan `nim` yang dilakukan pada baris ke-4 dan ke-5. Semuanya menggunakan *method* yang telah disediakan untuk melakukan akses terhadap atribut kelas.

Aturan ini berlaku secara umum di Java, dan sebaiknya diikuti, karena banyak *framework* yang dibangun menggunakan standar seperti ini, sehingga kedepannya, saat kita terbiasa dengan skema seperti ini, untuk melakukan integrasi dengan menggunakan *framework* di Java lebih mudah dan dapat dikerjakan dengan hasil

yang benar.

2.3.3 Overloading

Seperti dijelaskan sebelumnya, bahwa deklarasi konstruktor tidak terbatas pada sebuah deklarasi konstruktor saja, namun bisa lebih dari satu, konsep inilah yang dinamakan *overloading*, yaitu dimana sebuah *method* (karena konstruktor sebetulnya adalah sebuah *method*) yang dideklarasikan dengan nama yang sama, namun dengan beberapa perbedaan parameter.

Contoh kodenya pada kelas Mahasiswa adalah seperti berikut :

```
1  public class Mahasiswa {
2
3    private String nama;
4    private String nim;
5
6    public Mahasiswa() {
7      nama = "tidak ada";
8      nim = "00000";
9    }
10
11   public Mahasiswa(String nama, String nim) {
12     this.nim = nim;
13     this.nama = nama;
14   }
15
16   public void cetakInfo() {
17     System.out.println("Nama : " + nama);
18     System.out.println("NIM : " + nim);
19   }
20
21   public void setNama(String nama) {
```

```
22      this.nama = nama;
23    }
24
25    public String getNama() {
26      return nama;
27    }
28
29    public void setNim(String nim) {
30      this.nim = nim;
31    }
32
33    public String getNim() {
34      return nim;
35    }
36
37  }
```

Perhatikan blok baris ke-11 sampai baris ke-14, ini adalah contoh *overload* konstruktor untuk kelas `Mahasiswa`, dengan adanya deklarasi ini, maka akan kita coba manfaatkan pada kelas `Aplikasi` sehingga pembentukan instan kelas `Mahasiswa` jadi lebih ringkas. Berikut adalah contoh perubahan yang terjadi pada kelas `Aplikasi` :

```
1  public class Aplikasi {
2    public static void main(String args[]) {
3      Mahasiswa ami = new Mahasiswa("tamami", "19001");
4      Mahasiswa diva = new Mahasiswa("Diva", "19002");
5
6      ami.cetakInfo();
7      System.out.println();
8      diva.cetakInfo();
9    }
10 }
```

Silahkan lakukan *compile* ulang kemudian jalankan aplikasinya.

2.4 Kesimpulan

Bahwa secara implisit, konstruktor tanpa parameter akan terbentuk pada saat kita mendeklarasikan sebuah kelas, selain itu kita pun dapat membentuk lebih dari 1 (satu) konstruktor dengan perbedaan jumlah parameter yang kita sebut dengan *override*.

Implementasi yang dilakukan pada *field* atau atribut di dalam kelas adalah menjadikan aksesnya tertutup secara langsung untuk penggunaan dari luar kelas yang bersangkutan.

2.5 Tugas

Dari tugas pada Bab 1, kali ini kita modifikasi dengan mengubah konstruktor tanpa parameter untuk memberikan nilai *default* terhadap atribut *nama* dan *nomor anggota*.

Kemudian buat sebuah konstruktor lagi dengan parameter berupa *nama* dan *nomor anggota*.

Setelah kelas `Anggota` terbentuk, buat dua buah objek kemudian manfaatkan konstruktor dengan parameter untuk membentuk instan kelas bagi kedua objek tersebut.

Tampilkan isi informasi objeknya ke layar.

Bab 3

Struktur Percabangan dan Perulangan

3.1 Tujuan

Pada Bab ini diharapkan mahasiswa memahami konsep Percabangan dan Perulangan serta implementasinya pada bahasa pemrograman Java.

3.2 Pengantar

Kondisi percabangan adalah kondisi dimana alur dari logika program memiliki dua atau lebih pilihan yang harus dijalankan, msaing-masing pilihan akan mengakibatkan hasil yang berbeda. Istilah lain yang biasa disebut untuk mengungkapkan ini adalah seleksi.

Sedangkan kondisi perulangan adalah kondisi dimana suatu alur program perlu melakukan beberapa pekerjaan yang berulang untuk beberapa siklus tertentu.

Implementasi untuk kedua konsep tersebut mampu dilakukan dalam bahasa pemrograman apapun, namun kita akan mencoba mengimplementasikannya di ba-

hasa pemrograman Java.

3.3 Praktek

3.3.1 Percabangan

Percabangan di Java dapat dideklarasikan melalui beberapa cara, mari kita bahas macamnya satu satu.

Operator *Ternary*

Apabila kita memiliki kasus cabang yang sederhana, yang hasilnya dapat langsung dikembalikan dan disimpan dalam sebuah variabel, kita dapat menggunakan operator *ternary*. Format yang digunakan untuk dekalrasi operator *ternary* ini adalah seperti berikut :

```
(a) ? b : c
```

Keterangan dari kode tersebut adalah seperti berikut :

Huruf	Keterangan
a	seleksi yang hasilnya dapat bernilai `true` atau `false`
b	nilai yang dikembalikan apabila pernyataan pada huruf **a** bernilai `true` atau benar
c	nilai yang dikembalikan apabila pernyataan pada huruf **a** bernilai `false` atau salah

Perhatikan contoh kode berikut :

```
public class Aplikasi {
  public static void main(String args[]) {
    int a = 10;
    int b = 13;

```

```
6       String result = (a % 2 == 0) ? "bilangan genap" : "bilangan
        ganjil";
7       System.out.println(a + " adalah " + result);
8
9       result = (b % 2 == 0) ? "bilangan genap" : "bilangan ganjil";
10      System.out.println(b + " adalah " + result);
11    }
12  }
```

Perhatikan pada baris ke-6 dan ke-9, pada baris ini kita menggunakan operator *ternary* untuk melakukan seleksi sederhana apakah sebuah bilangan seperti `10` atau `13` pada variabel `a` dan `b` merupakan bilangan genap atau bilangan ganjil.

Dengan operator *ternary* kita tidak perlu menggunakan perintah `if` yang begitu panjang, cukup deklarasikan dalam satu baris kode, dengan cara mencari hasil sisa bagi dengan 2 (dua), apabila nilainya adalah 0 (nol), maka akan mengembalikan teks "bilangan genap", namun bila hasilnya tidak 0 (nol) maka akan mengembalikan teks "bilangan ganjil".

Blok Perintah `if`

Blok perintah `if` ini akan melakukan percabangan atau melakukan perintah yang berada dalam bloknya apabila pernyataan yang diberikan bernilai `true` atau benar.

Contoh kodenya adalah seperti berikut :

```
1 if(a) b;
```

Bila pernyataan pada bagian `a` bernilai `true`, maka pernyataan pada bagian `b` akan dijalankan, namun bila bernilai `false` maka pernyataan pada bagian `b` akan dilewati.

Bentuk lain dari perintah `if` apabila kita membutuhkan lebih banyak baris kode yang dieksekusi pada bagian `b` apabila pernyataan pada bagian `a` bernilai `true` adalah seperti berikut :

```
if(a) {
  b;
  c;
}
```

Sehingga apabila pernyataan pada bagian `a` bernilai `true`, maka pernyataan di dalam kurung kurawal (yaitu pada bagian `b` dan `c`) dapat dijalankan.

Contoh implementasi kodenya adalah seperti berikut :

```
public class Aplikasi {
  public static void main(String args[]) {
    int a = 10;
    if(a % 2 == 0) {
      System.out.println(a + " adalah bilangan genap");
      System.out.println("ini masih dari dalam struktur if");
    }
    System.out.println("akhir aplikasi");
  }
}
```

Perhatikan bahwa kode pada baris ke-5 dan ke-6 akan tercetak bila program dijalankan, namun bila nilai pada variabel `a` kita ubah menjadi bilangan ganjil, maka kedua baris tersebut akan dilewati, karena pernyataan pada parameter `if` tidak mengembalikan nilai `true`.

Blok Perintah `if...else`

Dengan menggunakan perintah `if`, maka apabila parameter yang diberikan bernilai `false`, maka aplikasi akan melewati begitu saja. Bagaimana bila nilai pada parameter `if` bernilai `false` namun kita tetap akan menangani hasilnya?

Solusi dari permasalahan tersebut ada pada blok perintah berikut :

```
if(a) b;
else c;
```

Seperti pada bahasan sebelumnya, bahwa bila bagian `a` bernilai `true` maka pernyataan pada bagian `b` akan dijalankan, namun bila hasil pada bagian `a` bernilai `false`, maka yang dijalankan adalah pernyataan pada bagian `c`.

Format lain apabila kita memerlukan lebih dari 1 (satu) baris perintah pada bagian `b` dan `c`, cukup berikan kurung kurawal untuk memberikan tanda blok yang dikerjakan, formatnya menjadi seperti berikut :

```
1 if(a) {
2   b;
3   c;
4 } else {
5   d;
6   e;
7 }
```

Namun pada contoh tersebut, apabila nilai pada bagian `a` bernilai `true` maka yang dikerjakan adalah pernyataan pada blok pertama, yaitu bagian `b` dan `c`, namun bila bernilai `false` maka yang dikerjakan adalah bagian `d` dan `e`.

Contoh implementasinya adalah seperti berikut :

```
1  public class Aplikasi {
2    public static void main(String args[]) {
3      int a = 13;
4
5      if(a % 2 == 0) {
6        System.out.println(a + " adalah bilangan genap");
7      } else {
8        System.out.println(a + " adalah bilangan ganjil");
9      }
10     System.out.println("akhir aplikasi");
11   }
12 }
```

Tentu saja hasil dari baris program di atas adalah tercetaknya `a` dengan keterangan `adalah bilangan ganjil`.

Selain bentuk sederhana seperti itu, perintah `if` dan `if...else...` pun sebetulnya bisa dibuat bertingkat, sehingga dapat melakukan percabangan atau seleksi beberapa kondisi dalam satu alur.

Blok perintah `switch...case`

Perintah `switch...case` ini digunakan apabila kita memiliki beberapa alternatif pilihan selain dalam bentuk `true` dan `false`.

Struktur perintah ini adalah seperti berikut :

```
switch(a) {
  case b:
    c;
    break;
  case d:
    e;
    break;
  default:
    f;
}
```

Dari kode di atas, yang akan dilakukan seleksi hasil adalah pada bagian `a`, apabila hasil pemeriksaan seleksi pada bagian `a` menghasilkan nilai `b`, maka yang akan dieksekusi adalah bagian `c`, sedangkan apabila hasil seleksi `a` merupakan nilai `d`, maka yang akan dijalankan adalah pada bagian `e`, terakhir apabila tidak ada satu nilai pun yang cocok, maka akan dijalankan blok kode yang berada pada bagian `f`.

Pilihan blok baris `default` sebetulnya adalah pilihan, boleh disertakan, atau tidak disertakan pun tidak apa-apa. Kemudian pemberian perintah `break` pada

tiap akhir `case` adalah karena apabila sebuah blok `case` dijalankan, maka setelah baris akhir dari `case` tersebut tidak ada `break`, maka akan dilanjutkan ke `case` berikutnya.

Contoh implementasinya adalah seperti kode berikut ini :

```
1  public class Aplikasi {
2    public static void main(String args[]) {
3      int pilihan = 2;
4
5      switch(pilihan) {
6        case 1:
7          System.out.println("Anda memilih angka 1");
8          break;
9        case 2:
10         System.out.println("Anda memilih angka 2");
11         break;
12       case 3:
13         System.out.println("Anda memilih angka 3");
14         break;
15       default:
16         System.out.println("Tidak ada pilihan");
17     }
18   }
19 }
```

Dari contoh diatas, isi variabel `pilihan` sudah kita tentukan terlebih dahulu, yaitu `2`, sehingga hasil keluarannya dapat kita tebak, yaitu menjalankan perintah pada baris ke-10.

Cobalah ganti isi variabel `pilihan` dengan angka lain, kemudian jalankan programnya.

3.3.2 Perulangan

Struktur perulangan pun dalam bahasa pemrograman Java memiliki beberapa macam bentuk, mari kita bahas apa saja bentuknya.

Blok perintah `for`

Struktur `for` ini membutuhkan 3 (tiga) parameter, formatnya adalah seperti berikut :

```
for(a; b; c) {
  d;
}
```

Pada bagian `a` akan berisi inisialisasi nilai yang akan dilakukan iterasi atau perulangan, pada bagian `b` merupakan pemeriksaan logika apakah iterasi akan dilanjutkan atau tidak, bila bernilai `true` maka akan dilanjutkan, bila `false` maka iterasi akan dihentikan.

Pada bagian `c` akan berisi *counter* yang akan dikerjakan di tiap akhir siklus masing-masing iterasi, sedangkan pada bagian `d` adalah kondisi atau pernyataan yang akan dijalankan di tiap siklus iterasi.

Contoh implementasi kodenya adalah seperti berikut :

```
public class Aplikasi {
  public static void main(String args[]) {
    for(int i=1; i<=5; i++) {
      System.out.println("data ke-" + i);
    }
  }
}
```

Pada kode di atas, kita melakukan inisiasi variabel `i` yang bertipe data *integer* dengan nilai `1`, kemudian prosesnya akan melakukan pemeriksaan logika dengan

pernyataan `i<=5`, bila hasilnya bernilai `true` maka proses berlanjut dengan menjalankan blok perintah yang ada di dalam kurung kurawal, bila `false` maka perintah yang berada di dalam kurung kurawal akan dilewati.

Setiap 1 (satu) siklus pengerjaan iterasi, maka perintah `i++` akan dijalankan diakhir siklus, kemudian kembali lagi ke pemeriksaan logika apakah hasilnya masih bernilai `true` atau `false`.

Blok perintah `do...while`

Blok perintah ini akan menjalankan minimal satu siklus iterasi, karena pemeriksaan logika untuk meneruskan atau menyudahi proses iterasi berikutnya berada di akhir siklus iterasi. Formatnya adalah seperti berikut :

```
1 do {
2   a;
3 } while(b);
```

Blok baris `a` adalah yang dikerjakan dalam sebuah siklus iterasi, dan pada bagian `b` adalah pemeriksa logika yang apabila bernilai `true`, maka siklus iterasi berikutnya dikerjakan, namun bila isinya bernilai `false` maka iterasi selesai.

Contoh implementasi di Java untuk jenis iterasi ini adalah seperti berikut :

```
1 public class Aplikasi {
2   public static void main(String args[]) {
3     int i = 1;
4     do {
5       System.out.println(i++);
6     } while(i < 6);
7   }
8 }
```

Kode di atas, pada baris ke-3 akan melakukan inisiasi nilai pada variabel `i` dengan angka `1`, kemudian siklus awal iterasi akan dikerjakan seperti pada baris

ke-5, yaitu mencetak isi dari variabel `i`, kemudian ada tanda `++` yang artinya setelah perintah pada baris ini dikerjakan, variabel `i` akan dijumlahkan dengan 1 (*increment*), setelah itu akan melakukan siklus iterasi berikutnya sampai nilai pada variabel `i` bernilai `6` yang artinya sudah tidak memenuhi persamaan pada baris ke-6, dengan kata lain bernilai `false`.

Blok perintah `while...`

Sama seperti bentuk iterasi sebelumnya, yaitu `do...while`, namun kali ini pemeriksaan logika yang menentukan apakah iterasi dikerjakan atau tidak ada di awal siklus iterasi. Bentuk blok perintahnya adalah seperti berikut :

```
1 while(a) {
2   b;
3 }
```

Pada bentuk di atas, pada bagian `a` akan diperiksa terlebih dahulu hasil operasi logikanya, bila bernilai `true`, maka perintah yang ada pada bagian `b` akan dikerjakan, namun bila bernilai `false` iterasi akan dilewatkan.

Contoh implementasi kodenya adalah seperti berikut :

```
1  public class Aplikasi {
2    public static void main(String args[]) {
3      int angka;
4      Scanner sc = new Scanner(System.in);
5
6      System.out.print("Masukkan angka : ");
7      angka = sc.nextInt();
8      int i=0;
9      while(i < angka) {
10       System.out.println("datanya : " + i++);
11     }
12   }
```

```
13 }
```

Kali ini kita memanfaatkan kelas `Scanner` untuk menerima masukkan dari pengguna, pada baris ke-4, kita siapkan instan dari kelas `Scanner` dengan sumber data dari input pengguna, dalam hal ini papan ketik.

Pada baris ke-6, kita memberikan informasi ke pengguna untuk memasukkan sebuah angka, yang kemudian pada baris ke-7 kita simpan angka yang telah dimasukkan oleh pengguna ke variabel `angka`.

Pada baris ke-8, kita buat variabel `i` dan kita isikan nilai awalnya adalah `0` (nol), pada blok baris ke-9 sampai ke-11, kita mulai melakukan pencetakan isi dari variabel `i` sampai nilainya sama dengan `angka` yang telah dimasukkan oleh pengguna.

3.4 Kesimpulan

Bahwa percabangan digunakan apabila kita ingin melakukan seleksi terhadap sebuah nilai, yang kemudian menentukan alur logika aplikasi yang dikerjakan. Sedangkan perulangan dapat kita gunakan apabila kita membutuhkan sebuah kode yang dijalankan berulang untuk beberapa siklus tertentu.

3.5 Tugas

Buatlah sebuah aplikasi sederhana, yang terdiri dari 3 (tiga) menu, judul menunya adalah seperti berikut :

1. Tambah
2. Kurang
3. Keluar

Menu ini akan terus berulang, sampai pengguna memilih atau memasukkan angka 3 (tiga).

Bila pengguna memilih angka 1 (satu) maka variabel yang telah disiapkan akan ditambahkan dengan 1 (satu) kemudian ditampilkan di layar, lalu menampilkan menu ini kembali.

Bila pengguna memilih angka 2 (dua), variabel yang telah disiapkan akan dikurangi dengan 1 (satu) kemudian ditampilkan di layar dan kembali memunculkan menu tersebut.

Bab 4

Paket

4.1 Tujuan

Pada Bab ini diharapkan mahasiswa memahami konsep dari paket (*package*) dan mampu mengimplementasikan konsep tersebut pada bahasa pemrograman Java.

4.2 Pengantar

Paket pada paradigma pemrograman berorientasi objek digunakan untuk mengelompokkan beberapa kelas yang mirip atau sejenis, bisa dianalogikan bahwa ini adalah sebuah kandar dengan nama berkas yang sejenis di dalamnya.

Fungsi paket yang lain adalah agar tidak terjadi deklarasi ambigu dari sebuah kelas, misalnya, bila kita ingin mendeklarasikan 2 (dua) atau lebih kelas dengan nama yang sama namun dengan tujuan atau fungsi yang berbeda, maka cukup menggunakan penamaan paket untuk membedakan bahwa kedua kelas tersebut memang berbeda secara fungsi.

Yang perlu dicatat adalah bahwa penamaan paket di Java akan mengikuti penamaan struktur kandar di *file system*, jadi nama paket akan mengikuti nama

kandar-nya.

4.3 Praktek

Kali ini akan kita coba implementasikan pengguna paket ini dan bagaimana cara memanfaatkan penamaan paket ini pada kelas `Aplikasi`.

Pertama kita perlu membuat kandar / *folder* dengan nama `data`. Nama kandar ini tentu saja harus sama dengan nama paket yang akan kita gunakan, karena Java mengikuti penamaan struktur kandar di *file system* yang kita gunakan.

Di dalam kandar `data`, kita membuat sebuah kelas dengan nama `Mahasiswa`, isi deklarasi kelasnya adalah seperti berikut :

```
1  package data;
2
3  public class Mahasiswa {
4    private String nama;
5    private String nim;
6
7    public Mahasiswa(String nim, String nama) {
8      this.nim = nim;
9      this.nama = nama;
10   }
11
12   public void cetak() {
13     System.out.println(nim + " : " + nama);
14   }
15 }
```

perhatikan deklarasi paket pada baris ke-1, penamaan paket ini mengikuti penamaan pada kandar yang telah kita buat sebelumnya, perhatikan besar kecilnya huruf karena ini berpengaruh.

Kelas tersebut hanya mendefinisikan 2 (dua) properti atau atribut `nim` dan `nama`, kemudian pada konstruktornya langsung ditetapkan 2 (dua) parameter untuk mengisi atribut itu, terakhir kita berikan *method* `cetak` untuk melakukan pencetakan isi dari atribut `nim` dan `nama` ke layar.

Selanjutnya, di luar kandar `data`, sejajar dengan kandar ini kita buat kelas `Aplikasi`. Isi dari berkas `Aplikasi.java` ini adalah seperti berikut :

```
1  import data.Mahasiswa;
2
3  public class Aplikasi {
4    public static void main(String args[]) {
5      Mahasiswa[] mhs = {
6        new Mahasiswa("19001", "tamami"),
7        new Mahasiswa("19002", "diva"),
8        new Mahasiswa("19003", "nabila")
9      };
10
11     for(Mahasiswa mahasiswa : mhs) {
12       mahasiswa.cetak();
13     }
14   }
15 }
```

Perhatikan baris pertamanya yang menggunakan perintah `import` untuk menyertakan kelas `Mahasiswa` pada kelas `Aplikasi`, apabila kelas `Mahasiswa` berada dalam satu kandar yang sama dengan kelas `Aplikasi`, penggunaan perintah `import` ini tidak perlu, berhubung letak kelas `Mahasiswa` berada dalam paket (kandar) `data`, maka kita perlu mendeklarasikan dengan perintah `import`.

Compile dan jalankan kode di atas sehingga hasil yang kita dapat seharusnya akan menampilkan 3 (tiga) data mahasiswa yang telah kita definisikan pada larik `mhs`.

4.4 Kesimpulan

Bahwa penggunaan paket dibutuhkan agar tidak ada definisi kelas yang konflik pada saat implementasi.

Deklarasi penamaan paket di bahasa pemrograman Java akan mengikuti struktur pada *file system*, sehingga perlu diperhatikan penamaan kandar pada *file system* yang digunakan.

4.5 Tugas

Dari tugas pada Bab 2, simpan kelas `Anggota` dalam paket `data`, kemudian tunjukkan cara memanggilnya, serta tunjukkan pula hasilnya.

Bab 5

Inheritance, Encapsulation, dan Polimorphism

5.1 Tujuan

Pada Bab ini diharapkan mahasiswa memahami konsep *inheritance* (pewarisan), *Encapsulation*, dan *Polimorphism*, serta bagaimana implementasi ketiga konsep tersebut pada bahasa pemrograman Java.

5.2 Pengantar

Konsep *inheritance* ini sering muncul dalam pembahasan sebuah paradigma pemrograman berorientasi objek, yaitu bagaimana sebuah kelas akan mewarisi atribut dan *method* milik kelas di atasnya, yang kemudian hanya tinggal menambahkan perilaku unik yang lebih detail daripada kelas yang mewarisi.

Dengan kata lain, kelas yang mewarisi, akan memiliki seluruh atribut dan *method* yang dideklarasikan pada kelas di atasnya.

Konsep *Encapsulation* adalah aturan pada paradigma pemrograman berori-

entasi objek bahwa seluruh informasi detail dalam kelas perlu disembunyikan, satu-satunya cara untuk melakukan akses terhadap informasi ini dilakukan melalui *interface* yang kita kenal dengan istilah *method* atau fungsi, atau prosedur.

Polimorfisme sendiri memiliki beberapa pengertian, yang pertama adalah *method* atau konstruktor yang memiliki banyak bentuk, sehingga mampu memproses objek berdasarkan tipe datanya, dalam pengertian lain yaitu sebuah *method* dengan implementasi yang bermacam-macam.

Bentuk implementasi dari polimorfisme ini sendiri nantinya dapat berupa *overloading* yang telah kita bahas sebelumnya, atau *overriding*.

Kita perjelas saja ketiga konsep tersebut di atas pada bagian praktek.

5.3 Praktek

5.3.1 Inheritance

Implementasi untuk *inheritance* atau pewarisan ini, misalkan kita memiliki sebuah kelas dengan nama `Personal` yang nantinya sebagai pewaris terhadap kelas `Mahasiswa`.

Deklarasi untuk kelas `Personal` ini kita simpan dalam paket `entitas`, berikut adalah isi kode dari kelas `Personal` :

```
1 package entitas;
2
3 public class Personal {
4
5   private String nik;
6   private String nama;
7
8   public Personal() {
9     nik = "3376000";
```

```
10      nama = "tidak ada";
11    }
12
13    public Personal(String nik, String nama) {
14      this.nik = nik;
15      this.nama = nama;
16    }
17
18    public void cetak() {
19      System.out.println(nik + " : " + nama);
20    }
21
22    // getter and setter
23    // ....
24
25  }
```

Kelas `Personal` ini memiliki 2 (dua) properti atau atribut dengan nama `nik` dan `nama`, memiliki 2 (dua) konstruktor, dan 5 (lima) buah *method*, yang 1 (satu) terlihat (yaitu *method* `cetak`), dan yang lain adalah *method* aksesor yang sengaja tidak disertakan karena terlalu makan banyak tempat.

Selanjutnya kita deklarasikan kelas `Mahasiswa` pada paket yang sama, yaitu paket `entitas`. Isi atau deklarasi kelasnya adalah seperti berikut :

```
1  package entitas;
2
3  public class Mahasiswa extends Personal {
4
5    private String nim;
6
7    public Mahasiswa(String nim) {
8      super();
9      this.nim = nim;
```

```
10   }
11
12   public Mahasiswa(String nim, String nik, String nama) {
13     super(nik, nama);
14     this.nim = nim;
15   }
16
17 }
```

Pada saat kita deklarasikan kelas `Mahasiswa` pada baris ke-3, kita melihat ada perintah baru, yaitu `extends`, perintah inilah yang digunakan untuk menunjukkan bahwa kelas `Mahasiswa` akan mewarisi atribut dan *method* milik kelas `Personal`.

Kemudian pada konstruktor `Mahasiswa` kita melihat ada perintah `super()` yang artinya sebetulnya adalah memanggil konstruktor tanpa parameter dari kelas `Personal`, yang tentunya secara otomatis seluruh deklarasi atribut dan *method* akan ditempelkan pada kelas `Mahasiswa` ini.

Sekarang kita perhatikan kondisi kelas `Aplikasi` yang nantinya akan membentuk objek dari kelas `Mahasiswa` ini, kelas `Aplikasi` akan dideklarasikan di luar paket `entitas`, deklarasinya adalah seperti berikut :

```
1 import entitas.Mahasiswa;
2
3 public class Aplikasi {
4   public static void main(String args[]) {
5     Mahasiswa[] mhs = {
6       new Mahasiswa("3376001", "19001", "tamami"),
7       new Mahasiswa("3376002", "19002", "diva"),
8       new Mahasiswa("3376003", "19003", "nabila")
9     };
10
11    for(Mahasiswa mahasiswa : mhs) {
12      mahasiswa.cetak();
```

```
13      }
14    }
15  }
```

Perhatikan bahwa pada baris ke-12, ada pemanggilan *method* `cetak`, padahal pada deklarasi milik kelas `Mahasiswa`, *method* tersebut sama sekali tidak ada. Hal ini karena kelas `Mahasiswa` sebetulnya mewarisi seluruh atribut dan *method* dari kelas `Personal`.

5.3.2 Encapsulation

Encapsulation atau enkapsulasi sendiri sebetulnya sudah kita implementasikan di bagian sebelumnya, kita pratinjau kembali kode dari kelas `Personal` berikut :

```
1  package entitas;
2
3  public class Personal {
4
5    private String nik;
6    private String nama;
7
8    public Personal() {
9      nik = "3376000";
10     nama = "tidak ada";
11   }
12
13   public Personal(String nik, String nama) {
14     this.nik = nik;
15     this.nama = nama;
16   }
17
18   public void cetak() {
19     System.out.println(nik + " : " + nama);
```

```
20    }
21
22    // getter and setter
23    // ...
24
25 }
```

Pada kelas tersebut, kita telah menyembunyikan atribut `nik` dan `nama`, dan memberikan *method* untuk melakukan akses terhadap kedua properti atau atribut tersebut dengan *method getter* dan *setter*.

Sampai disini kita telah mengimplementasikan konsep enkapsulasi dari paradigma pemrograman berorientasi objek dengan cara menyembunyikan detail dari kelas, dan hanya membuka *interface* yang dibutuhkan oleh kelas lain. Kedepan kita akan menggunakan pola ini sebagai standar.

5.3.3 Polimorphism

Seperti dijelaskan sebelumnya bahwa pada konsep *polimorphism* atau polimorfisme, konstruktor atau sebuah *method* mampu melakukan operasi yang berbeda-beda sesuai dengan tipe data objek yang akan di proses, atau sebuah *method* yang memiliki banyak implementasi berbeda.

Untuk definisi pertama sudah kita bahas pada Bab sebelumnya bahwa sebuah konstruktor, yang sebetulnya adalah *method* juga, dapat memiliki berbagai macam bentuk dengan jumlah parameter sebagai pembeda. Sedangkan untuk pengertian kedua, kita coba implementasikan seperti berikut.

Kita akan coba perluas implementasi kelas `Personal`, selain kelas `Mahasiswa`, kita akan membuat kelas `Dosen` yang merupakan pewaris dari kelas `Personal` juga, berikut isi dari kelas `Dosen` :

```
1 package entitas;
2
```

```
3  public class Dosen extends Personal {
4
5      private String nidn;
6
7      public Dosen(String nidn, String nama) {
8          super();
9          this.nidn = nidn;
10         setNama(nama);
11     }
12
13     public void cetak() {
14         System.out.println(nidn + " : " + getNama());
15     }
16
17 }
```

Lalu kita ubah deklarasi kelas `Mahasiswa` menjadi seperti berikut :

```
1  package entitas;
2
3  public class Mahasiswa extends Personal {
4
5      private String nim;
6
7      public Mahasiswa(String nim, String nama) {
8          super();
9          this.nim = nim;
10         setNama(nama);
11     }
12
13     public void cetak() {
14         System.out.println(nim + " : " + getNama());
15     }
16
```

```
}
```

Perhatikan bahwa masing-masing kelas, baik kelas `Mahasiswa` dan kelas `Dosen` memiliki `method cetak()`, sekarang kita lihat bagaimana kode pada kelas `Aplikasi` yang telah kita modifikasi seperti berikut :

```
import entitas.*;

public class Aplikasi {
  public static void main(String args[]) {
    Mahasiswa[] mhs = {
      new Mahasiswa("19001", "tamami"),
      new Mahasiswa("19002", "diva"),
      new Mahasiswa("19003", "nabila")
    };

    Dosen[] dosen = {
      new Dosen("1984001", "Dosen A"),
      new Dosen("1991001", "Dosen B"),
      new Dosen("1989002", "Dosen C")
    };

    System.out.println("Daftar Mahasiswa:");
    for(Personal personal : mhs) {
      personal.cetak();
    }

    System.out.println("\nDaftar Dosen:");
    for(Personal personal : dosen) {
      personal.cetak();
    }
  }
}
```

Pada baris ke-1, karena kita membutuhkan kelas `Mahasiswa` dan kelas `Dosen`, maka kita perlu melakukan `import`, kita menggunakan tanda `*` (bintang) disini hanya untuk meringkas bahwa seluruh kelas yang berada dalam paket `entitas` akan kita gunakan pada kelas ini, bentuk lain dengan tujuan yang sama dapat dideklarasikan seperti berikut :

```
1 import entitas.Mahasiswa;
2 import entitas.Dosen;
3 import entitas.Personal;
```

Pada blok baris ke-5 sampai ke-9, kita membuat sebuah larik dengan isi 3 (tiga) objek dari kelas `Mahasiswa`, kemudian pada blok baris ke-11 sampai ke-15, kita pun membuat sebuah larik dengan isi 3 (tiga) objek dari kelas `Dosen`.

Pada blok baris ke-18 sampai baris ke-20, dan baris ke-23 sampai ke-25, kita menggunakan kelas `Personal` untuk menampilkan datanya, namun objek yang diambil berbeda, yang satu dari kelas `Mahasiswa`, dan yang lain dari kelas `Dosen`.

Karena masing-masing kelas memiliki implementasi yang berbeda, maka hasil keluaran yang kita terima pun menjadi berbeda, inilah yang disebut polimorfisme, berikut hasil keluaran yang seharusnya tampil pada saat kita menjalankannya :

```
1 Daftar Mahasiswa:
2 19001 : tamami
3 19002 : diva
4 19003 : nabila
5
6 Daftar Dosen:
7 1984001 : Dosen A
8 1991001 : Dosen B
9 1989002 : Dosen C
```

5.4 Kesimpulan

Bahwa konsep pewarisan disediakan dalam paradigma pemrograman berorientasi objek untuk mempermudah pemrograman dengan cara menggunakan kelas yang sudah ada untuk kemudian dikembangkan kembali, jadi tidak perlu membangunnya dari awal.

Enkapsulasi ada pada paradigma pemrograman berorientasi objek untuk menjaga keamanan data, fleksibilitas, dan memudahkan pemeliharaan, jadi jangan sampai data pada properti terubah secara sembarangan tanpa terfilter melalui *interface* yang disediakan oleh si pembuat kelas dalam bentuk *method*.

Sedangkan adanya polimorfisme memungkinkan pemrogram mendefinisikan banyak konstruktor dan *method* untuk digunakan oleh pemrogram lain yang ingin menggunakan, di sisi lain, membuka peluang pemrogram lain untuk melakukan implementasi yang berbeda sesuai dengan kebutuhan.

5.5 Tugas

Pada sebuah tempat pelayanan penitipan hewan, akan dibuat sebuah aplikasi yang melakukan manajemen terhadap hewan yang dititipkan.

Tugas kali ini hanya membuat sebagian kecil dari bagian besar sebuah aplikasi tersebut, berikut yang perlu dikerjakan.

Buatlah sebuah kelas `Hewan`, kelas ini memiliki sebuah konstruktor dengan 2 (dua) buah parameter yang berisi nomor identitas dan nama pemilik dari `Hewan` tersebut, dan 2 (dua) buah *method*, yang pertama mengembalikan nilai berupa informasi dalam bentuk teks (String) dengan format {`id`} : {`pemilik`}, dan yang kedua akan mengembalikan nilai `true` atau `false` yang memberikan tanda bahwa hewan tersebut sedang dititipkan atau tidak, kita beri nama *method* ini sebagai `status()`.

Kemudian buat 2 (dua) kelas, `Anjing` dan `Ikan` yang mewarisi kelas `Hewan`, kelas `Anjing` akan memiliki atribut `statusSuntikRabies`, sedangkan kelas `Ikan` akan memiliki atribut `statusGantiAir`, masing-masing kelas tersebut akan merubah isi dari *method* `status` untuk menampilkan informasi apakah hewan tersebut dititipkan atau tidak, dan menampilkan informasi apakah sudah pernah disuntik rabies untuk `Anjing` dan apakah sudah pernah ganti air untuk `Ikan`.

Bab 6

Passing Object dan Overloading Methods

6.1 Tujuan

Pada Bab ini diharapkan mahasiswa memahami konsep *passing object* dan *overloading method* serta mampu mengimplementasikan konsep tersebut pada bahasa pemrograman Java.

6.2 Pengantar

Seperti terjemahan bebasnya bahwa *passing object* sebetulnya adalah melewatkan atau menyertakan atau mengirimkan sebuah objek untuk kemudian diproses atau digunakan dalam *method* yang membutuhkan.

Sedangkan *overload method* sebetulnya sudah pernah kita implementasikan pada kode yang kita ketik di Bab sebelumnya, yaitu bagaimana sebuah *method* dengan nama yang sama, namun dapat melakukan operasi yang berbeda tergantung tipe data pada parameter yang disertakan.

6.3 Praktek

6.3.1 Passing Object

Contoh paling sederhana untuk *passing object* ini adalah pada *method* aksesor seperti kode berikut :

```
1  public class Mahasiswa {
2    private String nim;
3    private String nama;
4
5    public Mahasiswa(String nim, String nama) {
6      this.nim = nim; this.nama = nama;
7    }
8
9    public String getNim() { return nim; }
10
11   public void setNim(String nim) { this.nim = nim; }
12
13   public String getNama() { return nama; }
14
15   public void setNama(String nama) { this.nama = nama; }
16 }
```

Pada kode di atas, konstruktor menyediakan 2 (dua) parameter `nim` dan `nama`, artinya kita dapat melewatkan 2 (dua) objek pada konstruktor ini dengan tipe data `String`, begitu pula dengan *method* `setNim()` dan `setNama()`, masing-masing memiliki sebuah parameter dengan tipe data `String`, sehingga kita dapat menyertakan sebuah objek dengan tipe data `String` ke dalamnya.

Kemudian untuk penggunaan *method* `getNim()` dan `getNama()`, *method* ini akan mengembalikan atau mengirimkan sebuah objek dengan tipe data `String`.

Mari kita lihat implementasinya pada kelas `Aplikasi` berikut :

```
1 public class Aplikasi {
2   public static void main(String args[]) {
3     Mahasiswa mhs = new Mahasiswa("19001", "tamami");
4     System.out.println(mhs.getNim() + " : " + mhs.getNama());
5   }
6 }
```

Kita lihat pada baris ke-3 bahwa pemanggilan konstruktor `Mahasiswa()` akan mengirimkan 2 (dua) buah objek bertipe data `String` dengan isi `"19001"` dan `"tamami"`. Kemudian pada baris ke-4, perintah `println` akan menerima objek yang dikirimkan dari *method* `getNim()` dan `getNama()`.

6.3.2 Overloading Method

Sama seperti konstruktor, sebuah *method* pun dapat dilakukan *overload* terhadapnya, perhatikan contoh kode berikut :

```
1  public class Mahasiswa {
2
3    private String nim;
4    private String nama;
5
6    public void setData(String nim, String nama) {
7      this.nim = nim;
8      this.nama = nama;
9    }
10
11   public void setData(int nim, String nama) {
12     this.nim = "" + nim;
13     this.nama = nama;
14   }
15 }
```

Pada contoh di atas, *method* `setData()` memiliki 2 (dua) bentuk, *method* ini mengalami *overloading* dimana *method* yang pertama akan menerima 2 (dua) parameter bertipe `String`, dan *method* yang kedua akan menerima sebuah parameter bertipe data `int` dan satu lagi bertipe `String`.

6.4 Kesimpulan

Bahwa *passing object* adalah istilah untuk mengirimkan sebuah objek ke dalam *method* atau konstruktor sebagai bahan untuk melakukan proses data, atau menjadi hasil dari sebuah proses di dalam *method*.

Sedangkan *overloading method* memberikan fleksibilitas kepada pemrogram untuk membuat sebuah *interface* kelas dengan nama yang sama namun mampu melakukan operasi terhadap berbagai objek dengan tipe data yang berbeda.

6.5 Tugas

Dari tugas pada Bab 4, tambahkan 2 (dua) buah *method* pada kelas `Anggota` dengan nama `setNoAnggota()` yang memiliki sebuah parameter, parameter pada *method* pertama akan bertipe `int` dan yang kedua akan bertipe `String`.

Implementasikan kedua *method* tersebut dengan cara memanggilnya dari kelas `Aplikasi`.

Bab 7

Rekursif, Static Modifier, dan Nested Classes

7.1 Tujuan

Pada Bab ini diharapkan mahasiswa memahami konsep dan mampu mengimplementasikan rekursif, *static modifier*, dan *nested classes* pada bahasa pemrograman Java.

7.2 Pengantar

Rekursif ini sebetulnya adalah sebuah *method* yang memanggil dirinya sendiri, yang dalam kondisi tertentu pemanggilan terhadap fungsi atau *method* tersebut berhenti.

Static modifier adalah satu kondisi dimana sebuah kelas, atribut, atau *method* akan berada pada satu alamat memori. Pada saat kita membuat sebuah objek, lalu objek tersebut kita bentuk instan dari sebuah kelas, pada saat itu pula kita menyiapkan sebuah ruang pada memori untuk objek tersebut menetap. Apabila

ada 2 (dua) objek yang kita bentuk, maka kita menyiapkan atau mengalokasikan sebesar 2 (dua) objek yang terbentuk, bayangkan saat ada ribuan objek yang terbentuk, otomatis penggunaan memori pun menjadi tidak efisien.

Ada saatnya kita melihat bahwa sebuah kelas, atau properti, atau *method* tidak perlu dicetak ke dalam sebuah objek, melainkan cukup dipanggil langsung dari posisi memori dia berada, implementasi langsung yang biasa dilakukan adalah pembuatan sebuah fungsi atau *method* yang berisi rumus yang akan digunakan dalam sebuah program, atau membuat sebuah konstanta yang nilainya tidak akan berubah dan digunakan dalam lingkup program, atau sebuah koneksi basis data yang tidak perlu dibuat berulang kali, cukup dibuatkan sebuah fungsi atau *method* untuk menangani koneksinya.

Berikutnya adalah *nested classes* yang sebetulnya ini adalah deklarasi kelas yang berada dalam kelas yang lain, kebutuhan pembentukan kelas ini biasanya spesifik hanya digunakan untuk kelas yang dideklarasikan secara publik.

7.3 Praktek

7.3.1 Rekursif

Implementasi paling mudah untuk melakukan rekursif ini adalah pada kasus bilangan faktorial, jadi bila ada bilangan `4!`, maka akan dihitung dengan rumus berikut :

$$4! = 4 * 3 * 2 * 1 = 24$$

Bila ada bilangan `5!`, maka akan dihitung seperti berikut :

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

Jadi bilangan akan dikalikan dengan bilangan tersebut yang telah dikurangi dengan 1 (satu), sampai nilai dari bilangan tersebut adalah 1 (satu).

Implementasi kode rekursif pada kasus faktorial ini adalah seperti berikut, kita membuat sebuah kelas `Matematika` dalam paket `util` dengan kode seperti ini :

```
1  package util;
2
3  public class Matematika {
4      public int faktorial(int n) {
5          if(n == 1) { return n; }
6          else {
7              return n * faktorial(n-1);
8          }
9      }
10 }
```

Pada baris ke-5, kita akan melakukan pemeriksaan bahwa apabila nilai `n` (parameter yang disertakan pada *method* `faktorial()`) bernilai `1` (satu), maka kembalikan nilai `1` (satu), namun selain itu akan dikembalikan hasil dari rumus pada baris ke-7, yaitu `n * faktorial(n-1)`, *method* `faktorial()` kembali di panggil namun kali ini dengan nilai `n-1`.

Pemanggilan fungsi atau *method* `faktorial()` ini akan terus berlanjut sampai nilai di `n` sama dengan `1`.

Kemudian kita melakukan implementasi penggunaan *method* `faktorial()` ini dalam kelas `Aplikasi` dengan kode seperti berikut :

```
1  import util.*;
2
3  public class Aplikasi {
4    public static void main(String args[]) {
5      Matematika mtk = new Matematika();
6      int hasil = mtk.faktorial(new Integer(args[0]));
7      System.out.println("hasil : " + hasil);
8    }
9  }
```

Seperti biasa kita membentuk sebuah instan dari kelas `Matematika()` pada baris ke-5, kemudian memanggil *method* `faktorial()` pada baris ke-6 yang parameternya diambilkan dari parameter aplikasi yang disertakan pada saat menjalankan program, kemudian pada baris ke-7, hasil faktorial yang telah diproses ditampilkan ke layar.

7.3.2 Static Modifier

Seperti penjelasan sebelumnya, penggunaan *static modifier* ini akan menyebabkan sebuah kelas, atribut, atau *method* menempati satu lokasi pada memori, untuk lebih jelasnya, mari perhatikan kelas berikut :

```
1 public class PercobaanStatic {
2     public static String atributStatik;
3 
4     public String atribut;
5 }
```

Disediakan 2 (dua) buah atribut, yang satu `static` dan yang satu tanpa `static`. Perhatikan saat kelas `PercobaanStatic` digunakan pada kelas `Aplikasi` berikut ini :

```
1 public class Aplikasi {
2   public static void main(String args[]) {
3     PercobaanStatic s1 = new PercobaanStatic();
4     PercobaanStatic s2 = new PercobaanStatic();
5 
6     s1.atribut = "Isi s1";
7     s2.atribut = "Isi s2";
8     s1.atributStatik = "isi statik s1";
9     s2.atributStatik = "isi statik s2";
10    System.out.println(s1.atribut);
11    System.out.println(s2.atribut);
```

```
12      System.out.println(s1.atributStatik);
13      System.out.println(s2.atributStatik);
14    }
15  }
```

Hasil dari kode tersebut apabila kita *compile* dan jalankan adalah seperti ini :

```
1 Isi s1
2 Isi s2
3 isi statik s2
4 isi statik s2
```

Yang janggal adalah hasil keluaran pada baris ke-3, isinya sama seperti pada baris ke-4, ini menunjukkan bahwa properti `atributStatik` berada pada satu alamat memori yang sama, sehingga apabila dari ada perubah, maka perubahan terakhir yang akan dihasilkan.

Beberapa IDE atau *editor* akan menghasilkan sebuah pesan bahwa properti `atributStatik` ini sebaiknya dipanggil dengan cara yang statik pula.

Jadi untuk pemberian nilainya sebetulnya bisa mengisikan nilai pada `s1.atributStatik`, atau `s2.atributStatik`, atau bahkan yang disarankan adalah bentuk `PercobaanStatik.atributStatik`.

7.3.3 Nested Classes

Contoh dari *nested classes* ini sebetulnya cukup mudah, yaitu ada sebuah kelas yang berada dalam kelas lainnya, perhatikan kode berikut yang diketik dalam berkas `Aplikasi.java` :

```
1 public class Aplikasi {
2
3   private Mahasiswa mhs;
4
5   public Aplikasi() { mhs = new Mahasiswa("19001", "tamami"); }
```

```
6
7   public Mahasiswa getMhs() { return mhs; }
8
9   public static void main(String args[]) {
10    new Aplikasi().getMhs().cetak();
11  }
12
13  class Mahasiswa {
14    private String nim;
15    private String nama;
16
17    public Mahasiswa(String nim, String nama) {
18      this.nim = nim; this.nama = nama;
19    }
20
21    public void cetak() {
22      System.out.println(nim + " : " + nama);
23    }
24  }
25 }
```

Perhatikan pada baris ke-13, deklarasi kelas `Mahasiswa` berada di dalam kelas `Aplikasi`, kemudian kelas aplikasi akan membuat sebuah objek `mhs` dari kelas `Mahasiswa` pada baris ke-3.

Kelas `Aplikasi` pun memiliki sebuah konstruktor yang akan membuat instan untuk objek `mhs` serta sebuah *method* untuk melakukan akses terhadap objek `mhs`.

Pada saat aplikasi dijalankan, maka akan berangkat dari baris ke-10, dimana akan dibuatkan instan objek secara anonim dari kelas `Aplikasi`, kemudian memanggil fungsi `getMhs()` untuk mendapatkan objek `mhs` yang berada di dalamnya, kemudian memanggil fungsi atau *method* `cetak()` milik objek `mhs`.

7.4 Kesimpulan

Bahwa penggunaan model rekursif dimungkinkan pada bahasa pemrograman Java, dimana sebuah *method* mampu memanggil dirinya sendiri, namun dengan catatan diberikan batasan yang jelas dimana proses rekursif harus berhenti, harap diperhatikan pula bahwa semakin banyak rekursif yang terjadi, maka penggunaan memori akan berbanding lurus dengan kejadiannya.

Kemudian ada saatnya dimana kita membutuhkan sebuah kelas, atribut, atau fungsi yang biasanya hanya berbentuk operasi atau rumusan dan tidak menyimpan sebuah data, sehingga setiap instan kelas tidak perlu membuatnya secara sendiri-sendiri, cukup deklarasikan sebagai `static` maka kelas, atau atribut, atau *method* tersebut cukup dialokasikan sekali pada suatu ruang di memori.

Sedangkan penggunaan *nested classes* tentunya akan spesifik pada kelas yang membutuhkan saja, dimana aksesnya terkadang akan dijadikan *private* sehingga hanya kelas yang berada di atasnya saja yang mampu melakukan akses terhadap kelas bersarang tersebut.

7.5 Tugas

Buatlah sebuah solusi rekursif untuk menghitung, apabila diberikan sebuah angka, maka akan menghitung penjumlahannya dari 1 sampai angka tersebut, bila angka 5 yang dimasukkan, maka akan menghitung seperti berikut :

$$5 = 5 + 4 + 3 + 2 + 1 = 15$$

apabila angka 7 yang dimasukkan, maka akan menghitung seperti berikut :

$$7 = 7 + 6 + 5 + 4 + 3 + 2 + 1 = 28$$

Mirip seperti faktorial, namun dalam operasi penjumlahan.

Bab 8

Sorting dan Searching

8.1 Tujuan

Pada Bab ini diharapkan mahasiswa memahami bagaimana implementasi *sorting* dan *searching* di Java.

8.2 Pengantar

Implementasi *searching* dan *sorting* pada bahasa pemrograman Java sebetulnya telah disediakan oleh sebuah kelas yang bernama `Collections`, mari kita manfaatkan kelas ini untuk melakukan *sorting* dan *searching*.

8.3 Praktek

8.3.1 Sorting

Yang pertama kita akan mengurutkan data yang sederhana, seperti teks, berikut contoh kodenya :

```
1 import java.util.*;
```

```
2
3 public class Aplikasi {
4   public static void main(String args[]) {
5     List<String> test = new LinkedList<>();
6
7     test.add("tamami");
8     test.add("diva");
9     test.add("nabila");
10
11    for(String data : test) {
12      System.out.println(data);
13    }
14
15    Collections.sort(test);
16    for(String data : test) {
17      System.out.println(data);
18    }
19  }
20 }
```

Pada baris ke-5, kita menyiapkan data dalam bentuk `List`, datanya yang dimasukkan berupa teks (`String`), kita melihat tampilan data apa adanya pada blok baris ke-11 sampai ke-13, kemudian pada baris ke-15, data yang berada dalam `List` diurutkan, hasil pengurutannya dapat kita lihat pada blok baris ke-16 sampai ke-18.

Lalu bagaimana saat bentuk yang akan kita urutkan berbentuk objek? Untuk hal ini perlakuan khusus, kita perlu mendeklarasikan sebuah kelas yang mengimplementasikan *interface* `Comparator` pada paket `java.util`.

Contoh kali ini kita akan menggunakan kelas `Mahasiswa` yang telah kita buat, kemudian kita implementasikan *interface* `Comparator` pada kelas `ComparatorMahasiswa`. Kelas ini akan mengimplementasikan pengurutan berda-

sarkan `nim`. Berikut deklarasi kelasnya :

```
1  package util;
2
3  import java.util.Comparator;
4  import entitas.Mahasiswa;
5
6  public class ComparatorMahasiswa implements Comparator<Mahasiswa> {
7
8      public int compare(Mahasiswa m1, Mahasiswa m2) {
9          if(new Integer(m1.getNim()) < new Integer(m2.getNim()))
       return -1;
10         else if(new Integer(m1.getNim()) == new Integer(m2.getNim()))
        return 0;
11         else return 1;
12     }
13
14 }
```

Perhatikan baris ke-6 bahwa kelas ini mengimplementasikan *interface* `Comparator`, yang perlu kita deklarasikan ulang adalah *method* `compare` milik *interface* `Comparator` dengan 2 (dua) parameter dengan tipe data `T` yang dapat dideklarasikan ulang menjadi sebuah kelas yang kita tentukan sendiri, dalam kasus ini kita tentukan sebagai kelas `Mahasiswa`.

Method `compare` ini akan mengikuti aturan seperti berikut, bila nilai `m1` kurang dari `m2`, maka akan mengembalikan nilai minus, apabila nilai `m1` sama dengan `m2`, maka akan mengembalikan nilai 0 (nol), sedangkan bila nilai `m1` lebih dari `m2` maka akan mengembalikan nilai positif.

Setelah *method* ini selesai kita deklarasikan, kita coba buat implementasi pada kelas `Aplikasi` seperti berikut :

```
1  import java.util.*;
2  import util.*;
```

```
3  import entitas.Mahasiswa;
4
5  public class Aplikasi {
6    public static void main(String args[]) {
7      List<Mahasiswa> test = new LinkedList<>();
8
9      test.add(new Mahasiswa("19003", "diva"));
10     test.add(new Mahasiswa("19002", "nabila"));
11     test.add(new Mahasiswa("19001", "tamami"));
12
13     System.out.println("sebelum sort:");
14     for(Mahasiswa data : test) {
15       System.out.println(data.getNim() + " : " + data.getNama());
16     }
17
18     System.out.println("\n\nSetelah sort:");
19     Collections.sort(test, new ComparatorMahasiswa());
20     for(Mahasiswa data : test) {
21       System.out.println(data.getNim() + " : " + data.getNama());
22     }
23   }
24 }
```

Pada bagian awal sama saja seperti sebelumnya, kita mengisikan data ke dalam `List` namun kali ini datanya berbentuk objek yang merupakan instan dari kelas `Mahasiswa`.

Kemudian pada baris ke-17, kita melihat pemanggilan *method* `sort` muncul parameter ke-2 yang merupakan instan dari kelas `ComparatorMahasiswa()` yang telah kita buat sebelumnya, gambaran hasil keluaran jika programnya kita jalankan adaalh seperti berikut :

```
1 Sebelum sort:
2 19003 : diva
```

```
19002 : nabila
19001 : tamami


Setelah sort:
19001 : tamami
19002 : nabila
19003 : diva
```

Hasilnya akan berurut sesuai dengan NIM (Nomor Induk Mahasiswa), apabila kita ingin mengurutkan berdasarkan namanya, kita tinggal modifikasi saja kelas `ComparatorMahasiswa`.

8.3.2 Searching

Banyak algoritma *searching* atau pencarian yang ada, namun pada kelas `Collections` hanya ada satu algoritma yang diimplementasikan, yaitu *binary search*.

Kondisi pencarian menggunakan algoritma ini mengharuskan kita mengurutkan datanya terlebih dahulu sebelum kemudian datanya dapat dicari.

Misalkan sebuah data akan berurut dari kiri ke kanan, pada bagian ujung kiri adalah data yang paling kecil, dan data pada ujung kanan adalah data yang paling besar, titik awal pencarian berada di tengah barisan data, apabila data yang dicari lebih besar dari data yang berada di tengah, maka pencarian akan berfokus pada bagian kanan, apabila data yang dicari lebih kecil dari data yang berada di tengah, maka pencarian akan fokus berada di bagian kiri baris data, terus membagi dua sampai datanya ditemukan.

Implementasinya adalah seperti berikut :

```
import java.util.*;

```

```
3  public class Aplikasi {
4    public static void main(String args[]) {
5      List<Integer> test = new LinkedList<>();
6
7      test.add(3);
8      test.add(2);
9      test.add(1);
10     test.add(5);
11     test.add(4);
12     test.add(7);
13     test.add(6);
14
15     System.out.println("Sebelum sort:");
16     for(Integer data : test) {
17       System.out.println("-> " + data);
18     }
19
20     System.out.println("\n\nSetelah sort:");
21     Collections.sort(test);
22     for(Integer data : test) {
23       System.out.println("=> " + data);
24     }
25
26     System.out.print("Hasil pencarian data 7 : ");
27     System.out.println(Collections.binarySearch(test, 7));
28   }
29 }
```

Perhatikan pada baris ke-21, kita perlu melakukan pengurutan data sebelum datanya dapat dicari dengan metode pencarian biner. Pencarian dilakukan pada baris ke-27.

Hasil keluaran dari program di atas adalah seperti berikut :

```
Sebelum sort:
-> 3
-> 2
-> 1
-> 5
-> 4
-> 7
-> 6


Setelah sort:
=> 1
=> 2
=> 3
=> 4
=> 5
=> 6
=> 7
Hasil pencarian data 7 : 6
```

Hasil pencarian data 7 adalah 6 karena penomoran indeks dari daftar akan dimulasi dari 0 (nol).

Lalu bagaimana untuk mencari sebuah objek dengan dengan cara di atas? kita perlu menyiapkan implementasi *interface* `Comparator` seperti sebelumnya.

Kelas `Mahasiswa` akan memiliki 3 (tiga) properti, yaitu `nik` dan `nama` yang diwariskan dari kelas `Personal`, dan `nim` yang dibentuk di kelas `Mahasiswa`.

Langkah pertama yang perlu kita ketahui adalah, kita akan mencari berdasarkan properti apa? Bila berdasarkan `nama`, maka data perlu diurutkan berdasarkan `nama` terlebih dahulu, apabila akan mencari berdasarkan properti `nim`, maka `nim` perlu diurutkan terlebih dahulu.

Contoh berikut adalah kita mencari sebuah objek berdasarkan `nim`, maka kita

perlu mengurutkan data berdasarkan `nim` terlebih dahulu, berikut isi kelas aplikasinya :

```
1  import java.util.*;
2  import util.ComparatorMahasiswa;
3  import entitas.Mahasiswa;
4
5  public class Aplikasi {
6    public static void main(String args[]) {
7      List<Mahasiswa> data = new LinkedList<>();
8
9      data.add(new Mahasiswa("19008", "tamami"));
10     data.add(new Mahasiswa("19005", "diva"));
11     data.add(new Mahasiswa("19003", "nabila"));
12     data.add(new Mahasiswa("19001", "peni"));
13     data.add(new Mahasiswa("19002", "honda"));
14     data.add(new Mahasiswa("19004", "suzuki"));
15     data.add(new Mahasiswa("19006", "mazda"));
16     data.add(new Mahasiswa("19007", "toyota"));
17
18     System.out.println("Sebelum sort:");
19     for(Mahasiswa mhs : data) {
20       System.out.println("-> " + mhs.getNim() + " : " + mhs.getNama()
     );
21     }
22
23     System.out.println("\n\nSetelah sort:");
24     Collections.sort(data, new ComparatorMahasiswa());
25     for(Mahasiswa mhs : data) {
26       System.out.println("=> " + mhs.getNim() + " : " + mhs.getNama()
     );
27     }
28
```

```
29      System.out.print("Hasil pencarian data mahasiswa untuk nim 19004
        : ");
30      System.out.println(Collections.binarySearch(data, new Mahasiswa
        ("19004", null), new ComparatorMahasiswa()));
31    }
32 }
```

Perhatikan baris ke-24, pada saat mengurutkan data, kita menggunakan instan dari kelas `ComparatorMahasiswa`, dengan kelas ini pula nantinya kita akan mencari data.

Kemudian perhatikan pada baris ke-30 bahwa kita akan mencari data pada larik `data`, data yang kita cari adalah objek dari kelas `Mahasiswa` dengan `nim` berisi `19004`, kelas `ComparatorMahasiswa` kita tempatkan pada parameter ke-3 dari *method* `binarySearch` ini. Sekarang perhatikan deklarasi kelas `ComparatorMahasiswa` berikut :

```
1  package util;
2
3  import java.util.Comparator;
4  import entitas.Mahasiswa;
5
6  public class ComparatorMahasiswa implements Comparator<Mahasiswa> {
7
8      public int compare(Mahasiswa m1, Mahasiswa m2) {
9          return m1.getNim().compareTo(m2.getNim());
10     }
11
12 }
```

perhatikan baris ke-6, bahwa kelas ini akan mewarisi *interface* `Comparator`, dimana kelas `ComparatorMahasiswa` perlu mengimplementasikan *method* `compare` milik *interface* `Comparator`.

Method `compare` ini seperti pembahasan sebelumnya, perlu mengembalikan nilai negatif bila parameter `m1` lebih kecil dari `m2`, mengembalikan nilai positif apabila kebalikan dari itu, atau mengembalikan nilai 0 (nol) yang artinya `m1` sama dengan `m2`.

Kita menggunakan *method* bawaan kelas `String` untuk melakukan komparasi itu, nama *method*-nya adalah `compareTo()`.

Hasil keluaran dari kode `Aplikasi` tersebut akan terlihat seperi berikut :

```
Sebelum sort:
-> 19008 : tamami
-> 19005 : diva
-> 19003 : nabila
-> 19001 : peni
-> 19002 : honda
-> 19004 : suzuki
-> 19006 : mazda
-> 19007 : toyota


Setelah sort:
=> 19001 : peni
=> 19002 : honda
=> 19003 : nabila
=> 19004 : suzuki
=> 19005 : diva
=> 19006 : mazda
=> 19007 : toyota
Hasil pencarian data mahasiswa untuk nim 19004 : 3
```

Ini artinya apabila datanya ditemukan, nilai angka yang dikembalikan adalah nomor indeks data dengan posisi data awal adalah 0 (nol).

8.4 Kesimpulan

Bahwa kita dapat memanfaatkan fitur pengurutan dan pencarian dari kelas `Collections` yang telah disediakan oleh Java.

Untuk mengurutkan data dalam bentuk objek, kita perlu membuat sebuah kelas yang mengimplementasikan *method* `compare` milik *interface* `Comparator`, di *method* inilah kita menentukan berdasarkan apa data akan diurutkan.

Kemudian untuk melakukan pencarian pun, kita dapat menggunakan metode pencarian biner yang telah disediakan kelas `Collections` di Java, namun barisan data yang ingin kita cari darinya perlu kita urutkan terlebih dahulu.

Kelas yang mengimplementasikan `Comparator` sebaiknya adalah kelas yang sama yang ditujukan untuk melakukan pengurutan data, ataupun pencarian data.

8.5 Tugas

Buatlah sebuah kelas yang mengimplementasikan `Comparator`, namun kali ini akan digunakan untuk mengurutkan dan mencarikan berdasarkan **nama**.

Bab 9

Exception dan Debugging

9.1 Tujuan

Pada Bab ini diharapkan mahasiswa mampu memahami pengertian dan bentuk *exception* pada Java dan mampu melakukan *debugging* pada bahasa pemrograman Java.

9.2 Pengantar

Exception sendiri sebetulnya adalah suatu kondisi dimana program menemukan kesalahan yang tidak semestinya saat instruksinya dijalankan.

Jadi pada saat kita *compile* kode program yang telah kita bangun, *compiler* tidak menemukan kesalahan ketikkan atau logika kode program, namun pada saat aplikasi dijalankan, semua fungsinya diujicoba, barulah muncul suatu kesalahan yang tidak semestinya, kondisi inilah yang disebut *exception*.

Sedangkan *debugging* adalah sebuah cara untuk mencari dan mengurangi kesalahan atau kerusakan dari kode program yang telah dibangun.

9.3 Praktek

Pada praktek sebelumnya mungkin ada yang sudah pernah mencoba kode berikut :

```
1 public class Aplikasi {
2   public static void main(String args[]) {
3     int angka = new Integer(args[0]);
4
5     System.out.println(angka);
6   }
7 }
```

Kode tersebut, apabila dilakukan *compile* tidak akan ada masalah. Masalah timbul apabila aplikasi dijalankan tanpa parameter seperti kode berikut :

```
1 $ java Aplikasi
```

Perhatikan bahwa sebuah *exception* muncul akibat perintah tersebut, yang memberikan pesan ke kita bahwa akses ke larik `args` melewati batas (*out of bounds*), karena kita melakukan akses pada baris ke-3 untuk mengambil data larik `args` yang pertama.

Informasi *exception* semacam ini akan sering timbul pada program yang kita bangun yang biasanya bersumber dari desain yang kurang lengkap. Namun seiring dengan jam terbang pemrogram, permasalahan seperti ini biasanya akan cepat teratasi.

Untuk *exception* sendiri sebetulnya dapat kita produksi sendiri dari logika kode program yang kita bangun, kita akan coba memproduksi *exception* dimana dari kelas `Mahasiswa` yang sudah terbentuk objeknya, kita akan seleksi bahwa data `nim` untuk objek dari kelas `Mahasiswa` harus sudah terisi, bila belum, kita lemparkan sebagai *exception*.

Kode pada kelas `Mahasiswa` yang sudah terbentuk tidak ada perubahan, kita

akan membuat implementasinya pada kelas `Aplikasi` berikut :

```
1 import entitas.Mahasiswa;
2
3 public class Aplikasi {
4   public static void main(String args[]) throws Exception {
5     Mahasiswa mhs1 = new Mahasiswa();
6
7     if(mhs1.getNim().equals("19000")) {
8       throw new Exception("Data NIM belum ada");
9     }
10  }
11 }
```

Pada kode di atas terlihat bahwa kelas `Mahasiswa` yang digunakan berasal dari paket `entitas`, perbedaan lain yang kita lihat adalah adanya pernyataan `throws Exception` pada baris ke-4, pernyataan ini berhubungan dengan baris ke-8, yang apabila nilai dari `nim` milik kelas `Mahasiswa` bernilai `19000`, artinya belum diisikan, masih berupa nilai *default*, maka kita lemparkan sebagai sebuah *exception* dengan pesan `Data NIM belum ada`.

```
1 MacBook-Pro:test tamami$ java Aplikasi
2 Exception in thread "main" java.lang.Exception: Data NIM belum ada
3         at Aplikasi.main(Aplikasi.java:9)
4 MacBook-Pro:test tamami$
```

Dari hasil keluaran di atas, *debugging*-nya cukup mudah, kita tinggal perhatikan pada berkas `Aplikasi.java` pada baris ke-9, kesalahan ini muncul karena belum diisikan, tinggal ubah saja kodenya menjadi seperti berikut :

```
1 import entitas.Mahasiswa;
2
3 public class Aplikasi {
4   public static void main(String args[]) throws Exception {
5     Mahasiswa mhs1 = new Mahasiswa();
```

```
6       mhs1.setNim("19001");
7
8       if(mhs1.getNim().equals("19000")) {
9         throw new Exception("Data NIM belum ada");
10      }
11
12      mhs1.cetak();
13    }
14  }
```

Pada baris ke-6, kita mengisikan `nim` dengan `19001`, dan pada baris ke-12 kita mencoba mencetak isinya. Hasilnya, *exception* tersebut sudah tidak dikeluarkan lagi, dan yang tercetak adalah informasi dari objek `mhs1`.

Kita pun dapat membangun sebuah kelas *Exception* yang akan menampilkan pesan yang kita rancang sendiri dengan cara mewarisi kelas `Exception` kemudian membuat konstruktornya, berikut contoh kodenya :

```
1 package exceptions;
2
3 public class ErrorNimException extends Exception {
4
5     public ErrorNimException(String message) {
6         super(message);
7     }
8
9 }
```

Kita buat dengan nama `ErrorNimException` di dalam paket `exceptions`, kita membuat satu konstruktor saja dengan parameter `message` di dalamnya, untuk menyimpan detail kesalahannya apa.

Pada kelas `Aplikasi` pun kita ubah agar menggunakan kelas `ErrorNimException` yang telah kita buat, berikut kodenya :

```
1 import entitas.Mahasiswa;
2 import exceptions.ErrorNimException;
3
4 public class Aplikasi {
5   public static void main(String args[]) throws Exception {
6     Mahasiswa mhs1 = new Mahasiswa();
7
8     if(mhs1.getNim().equals("19000")) {
9       throw new ErrorNimException("Data NIM belum ada");
10    }
11
12    mhs1.cetak();
13  }
14 }
```

Pada baris ke-9 kita sudah menggunakan *exception* yang kita bangun sendiri, hasil keluarannya akan terlihat seperti berikut :

```
1 MacBook-Pro:test tamami$ java Aplikasi
2 Exception in thread "main" exceptions.ErrorNimException: Data NIM
    belum ada
3         at Aplikasi.main(Aplikasi.java:9)
4 MacBook-Pro:test tamami$
```

Perhatikan bahwa *exception* akan menghentikan proses aplikasi sehingga perintah `cetak` pada baris ke-12 tidak akan dieksekusi. Kesalahan ini muncul pada berkas `Aplikasi.java` pada baris ke-9, cara menyelesaikan *bug* ini hanya tinggal mengisikan `nim` saja, berikut solusi kodenya :

```
1 import entitas.Mahasiswa;
2 import exceptions.ErrorNimException;
3
4 public class Aplikasi {
5   public static void main(String args[]) throws Exception {
6     Mahasiswa mhs1 = new Mahasiswa();
```

```
7       mhs1.setNim("19001");
8       mhs1.setNama("tamami");
9
10      if(mhs1.getNim().equals("19000")) {
11        throw new ErrorNimException("Data NIM belum ada");
12      }
13
14      mhs1.cetak();
15    }
16  }
```

Hasil keluaran dari kode tersebut akan terlihat seperti berikut :

```
1 MacBook-Pro:test tamami$ java Aplikasi
2 19001 : tamami
3 MacBook-Pro:test tamami$
```

9.4 Kesimpulan

Bahwa *exception* muncul karena adanya kesalahan pada saat *runtime*, pada kode program tidak ada kesalahan ketik, namun pada saat berjalannya aplikasi, ada kesalahan-kesalahan yang tidak dapat diproses oleh program sehingga muncul *exception*.

Munculnya *exception* akan menghentikan proses aplikasi sampai kondisi kode diperbaiki.

Exception pun dapat kita bangun sendiri yang spesifik sesuai kebutuhan aplikasi yang kita bangun.

9.5 Tugas

Perhatikan kelas `Mahasiswa` berikut :

```
1  package entitas;
2  public class Mahasiswa extends Personal {
3
4      private String nim;
5
6      public Mahasiswa() {
7          nim = "19000";
8          setNama("tidak ada");
9      }
10
11     public Mahasiswa(String nim, String nama) {
12         super();
13         this.nim = nim;
14         setNama(nama);
15     }
16
17     public void cetak() {
18         System.out.println(nim + " : " + getNama());
19     }
20
21     /**
22      * @return the nim
23      */
24     public String getNim() {
25         return nim;
26     }
27
28     /**
29      * @param nim the nim to set
30      */
31     public void setNim(String nim) {
32         if(nim.equals("19000")) {
33
```

```
34          }
35          this.nim = nim;
36      }
37
38  }
```

Buatlah sebuah *exception* `ErrorNamaException` untuk diimplementasikan pada kode berikut :

```
1  import entitas.Mahasiswa;
2  import exceptions.ErrorNamaException;
3
4  public class Aplikasi {
5    public static void main(String args[]) throws Exception {
6      Mahasiswa mhs1 = new Mahasiswa();
7      mhs1.setNim("19001");
8
9      if(mhs1.getNama().equals("tidak ada")) {
10       throw new ErrorNamaException("Data Nama belum ada");
11     }
12
13     mhs1.cetak();
14   }
15 }
```

Lalu perbaiki kode di atas agar kesalahan (*exception*) tersebut tidak muncul.

Bab 10

Handling Errors

10.1 Tujuan

Pada Bab ini diharapkan mahasiswa memahami dan mampu menangani kesalahan yang muncul pada aplikasi yang dibangun menggunakan bahasa pemrograman Java.

10.2 Pengantar

Pada bab sebelumnya kita telah membahas bagaimana sebuah *exception* terproduksi, dan bagaimana cara memperbaiki dengan mengubah kodenya. Adakalanya sebuah kesalahan yang muncul tidak memerlukan perbaikan kode, namun cukup perbaiki data yang diperlukan pada saat proses datanya terjadi.

Pada Bab ini kita akan coba menangkap sebuah *exception* dan menjadikan sebuah informasi pada pengguna untuk memperbaiki *input* datanya agar aplikasi atau program dapat berjalan sebagaimana mestinya.

10.3 Praktek

Untuk menangkap sebuah *exception* kita dapat menggunakan blok perintah berikut :

```
try {
...  // blok pertama
} catch(Exception e) {
... // blok kedua
}
```

Setiap perintah yang nantinya dapat menghasilkan *exception* akan kita tempatkan di blok pertama dari kode di atas, kemudian saat *exception* muncul, kita akan melakukan prosesnya di dalam blok kedua.

Bentuk lain adalah seperti berikut ini :

```
try {
... // blok pertama
} catch(Exception e) {
... // blok kedua
} finally {
... // blok ketiga
}
```

Perbedaannya hanya pada pernyataan `finally`, dimana dalam kondisi apapun, pernyataan pada blok `finally` akan selalu di proses.

Perhatikan contoh kode berikut :

```
import entitas.Mahasiswa;

public class Aplikasi {
  public static void main(String args[]) throws Exception {
    Mahasiswa mhs1 = new Mahasiswa();
    try {
      mhs1.setNim(args[0]);
```

```
8         mhs1.setNama(args[1]);
9       } catch(Exception e) {
10        System.err.println("ada kesalahan nim/nama tidak disertakan di
      parameter");
11        System.out.println("Gunakan : java Aplikasi {nim} {nama}");
12      }
13
14      mhs1.cetak();
15    }
16 }
```

Kita membuat sebuah blok `try...catch` pada baris ke-6 sampai ke-13, kita akan melakukan pengisian nilai atribut `nim` dan `nama` dari parameter aplikasi pada baris ke-7 dan ke-8, apabila pengguna tidak memberikan parameter apa pun pada aplikasi, maka blok `catch` akan dijalankan, yaitu menampilkan informasi kesalahan dan bagaimana cara menggunakan perintah aplikasi.

Berbeda dengan tanpa pemberian blok `try...catch`, aplikasi akan berhenti ketika bertemu dengan kasus yang sama, namun karena kita telah menangkap `exception` pada blok `try...catch`, Java menganggap kita sudah menangani *exception* tersebut dan aplikasi berjalan sebagaimana biasanya.

Coba jalankan kode tersebut dan lihat bagaimana hasilnya.

Pemberian pernyataan `finally` memungkinkan kita memberikan alternatif lain apabila sebuah *exception* tertangkap, atau ada proses lain yang harus dilengkapi setelah proses pada blok `try...catch` selesai dieksekusi. Perhatikan contoh kode berikut :

```
1 import entitas.Mahasiswa;
2
3 public class Aplikasi {
4   public static void main(String args[]) throws Exception {
5     Mahasiswa mhs1 = new Mahasiswa();
```

```
6       try {
7         mhs1.setNim(args[0]);
8         mhs1.setNama(args[1]);
9       } catch(Exception e) {
10        System.err.println("ada kesalahan nim/nama tidak disertakan di
      parameter");
11        System.out.println("Gunakan : java Aplikasi {nim} {nama}");
12      } finally {
13        if(mhs1.getNim().equals("19000")) {
14          mhs1.setNim("19001");
15          mhs1.setNama("na");
16        }
17      }
18
19      mhs1.cetak();
20    }
21  }
```

Perhatikan pada blok `finally`, saat data pada objek `mhs1` berupa data *default* dari konstruktor, maka kita mengubahnya dengan mengisikan `nim` menjadi `19001` dan `nama` menjadi `na`, perhatikan hasil keluaran program yang berbeda dengan sebelumnya.

10.4 Kesimpulan

Dengan menggunakan *exception handling*, aplikasi atau program yang dibuat menjadi lebih bersih, karena kode kesalahan yang dikeluarkan saat kesalahan terjadi tidak lagi serumit sebelumnya, pesan yang dikeluarkan lebih humanis dan diharapkan dapat dimengerti oleh pengguna aplikasi.

10.5 Tugas

Pada kelas `Mahasiswa`, berikan pengecekan pada saat melakukan pengisian data pada `nim`, dimana data yang masuk harus berisi angka (dapat dilakukan dengan `new Integer()` atau `Integer.parseInt()`), hasil dari kondisi *exception handling* dilemparkan ke luar agar dapat ditangkap pada kelas yang menggunakan.

Kemudian buat sebuah kelas `Aplikasi` yang memanfaatkan kelas `Mahasiswa`, dan isikan data `nim` yang mengandung karakter selain angka, tangkap *exception* ini dan berikan informasi kepada pengguna bahwa data `nim` yang diberikan harus berupa angka.

Bab 11

Image, Audio, dan Animasi

11.1 Tujuan

Pada Bab ini diharapkan mahasiswa memahami bagaimana menggunakan pustaka yang dapat melakukan akses terhadap *image*, *audio*, dan animasi.

11.2 Pengantar

Ada saatnya kita ingin menampilkan sebuah gambar pada aplikasi yang kita bangun, mungkin untuk menampilkan foto profil dari pengguna, atau gambar dari sebuah katalog, pada bagian ini kita akan coba bagaimana melakukan akses terhadap gambar untuk kemudian ditampilkan pada sebuah aplikasi berbasis GUI (*Graphical User Interface*).

Untuk saat ini, kita hanya dapat menampilkan gambar dengan tipe data atau ekstensi jpg, gif, atau png.

Untuk audio sendiri pun kita akan coba pada bab ini, bagaimana dengan bahasa pemrograman Java dan pustaka yang telah disediakan mampu untuk memainkan berkas audio dengan format data atau ekstensi AIFC, AIFF, AU, SND, atau

WAVE.

Pada bab ini pula kita akan mencoba membuat sebuah animasi sederhana melalui sudut pandang pemrograman berorientasi objek yang mampu Java lakukan.

11.3 Praktek

11.3.1 Image

Untuk memunculkan sebuah gambar pada jendela tentunya kita perlu untuk membuat jendelanya terlebih dahulu, di dalam jendela yang kita buat, kita akan masukkan komponen yang mampu menampilkan sebuah gambar yang kita pilih.

Perhatikan kode berikut :

```
1  import javax.swing.JFrame;
2  import javax.swing.ImageIcon;
3  import javax.swing.JLabel;
4
5  public class Aplikasi {
6    public static void main(String args[]) {
7      JFrame frame = new JFrame();
8    ImageIcon icon = new ImageIcon("img/foto-profile.jpeg");
9    JLabel label = new JLabel(icon);
10   frame.add(label);
11   frame.setDefaultCloseOperation
12           (JFrame.EXIT_ON_CLOSE);
13   frame.pack();
14   frame.setVisible(true);
15   }
16 }
```

Perhatikan baris ke-7 dimana kita membuat sebuah instan dari kelas `JFrame`, ini dimaksudkan kita akan membuat sebuah jendela utama dari aplikasi kita.

Pada baris ke-8, kita memanfaatkan kelas `ImageIcon` untuk menampung dan menampilkan gambar yang kita pilih. Parameter yang dibutuhkan oleh kelas ini hanyalah sebuah alamat letak berkas gambar tersimpan. Pastikan bahwa alamat dan nama berkas yang tertera di dalam ini benar.

Kemudian agar gambar dapat ditampilkan jendela `frame`, maka kita membutuhkan komponen `JLabel` yang akan menampung objek dari `ImageIcon` dan menampilkannya di layar, seperti deklarasi pada baris ke-9.

Kemudian pada baris ke-10, objek `label` yang berisi gambar dalam instan kelas `ImageIcon`, dimasukkan ke dalam jendela `frame`.

Baris ke-11 dan ke-12 adalah pelengkap kode dimana pada baris ke-11 untuk menjadikan tombol silang atau merah yang ada di pojok kanan atas atau pojok kiri atas jendela berfungsi untuk keluar dari jendela dan menghentikan aplikasi. Kemudian pada baris ke-12 digunakan agar ukuran jendela `frame` menyesuaikan ukuran isi di dalamnya.

Pada baris ke-14 kita memerintahkan jendela `frame` untuk muncul ke layar.

11.3.2 Audio

Untuk menjalankan berkas audio, berkas yang dapat didukung oleh Java adalah berkas dengan format AIFC, AIFF, AU, SND, dan WAVE.

Kali ini kita akan mencoba menjalankan berkas dengan format AU dengan menggunakan kelas `Clip`. Kodenya akan kita pisahkan menjadi 2 (dua) bagian (kelas), kelas pertama akan melakukan tugasnya sebagai *player* sedangkan kelas kedua akan melakukan tugasnya sebagai pusat kontrol aplikasi.

Kita buat kelas yang pertama terlebih dahulu, kodenya akan terlihat seperti berikut :

```
1 package audio;
2
```

```
3 import java.util.*;
4 import java.io.*;
5 import javax.sound.sampled.*;
6
7 public class MyPlayer {
8   private Long currentFrame;
9   private Clip clip;
10  private String status;
11  private AudioInputStream audioInputStream;
12  private String filePath;
13
14  public MyPlayer(String filePath) throws
     UnsupportedAudioFileException, IOException,
     LineUnavailableException {
15    audioInputStream = AudioSystem.getInputStream(new File(filePath).
     getAbsoluteFile());
16    clip = AudioSystem.getClip();
17    clip.open(audioInputStream);
18    clip.loop(Clip.LOOP_CONTINUOUSLY);
19    this.filePath = filePath;
20  }
21
22  public void play() {
23    clip.start();
24    status = "play";
25  }
26
27  public void pause() {
28    if(status.equals("paused")) {
29      System.out.println("player sudah dalam kondisi pause");
30      return;
31    }
32    this.currentFrame = this.clip.getMicrosecondPosition();
```

```
33      clip.stop();
34      status = "paused";
35    }
36
37    public void resumeAudio() throws UnsupportedAudioFileException,
       IOException, LineUnavailableException {
38      if(status.equals("play")) {
39        System.out.println("sedang memutar lagu");
40        return;
41      }
42      clip.close();
43      resetAudioStream();
44      clip.setMicrosecondPosition(currentFrame);
45      this.play();
46    }
47
48    public void restart() throws UnsupportedAudioFileException,
       IOException, LineUnavailableException {
49      clip.stop();
50      clip.close();
51      resetAudioStream();
52      currentFrame = 0L;
53      clip.setMicrosecondPosition(0);
54      this.play();
55    }
56
57    public void stop() throws UnsupportedAudioFileException,
       IOException, LineUnavailableException {
58      currentFrame = 0L;
59      clip.stop();
60      clip.close();
61    }
62
```

```
63   public void jump(long c) throws UnsupportedAudioFileException,
      IOException, LineUnavailableException {
64     if(c > 0 && c < clip.getMicrosecondLength()) {
65       clip.stop();
66       clip.close();
67       resetAudioStream();
68       currentFrame = c;
69       clip.setMicrosecondPosition(currentFrame);
70       this.play();
71     }
72   }
73
74   public void resetAudioStream() throws UnsupportedAudioFileException
       , IOException, LineUnavailableException {
75     clip.open(audioInputStream);
76     clip.loop(Clip.LOOP_CONTINUOUSLY);
77   }
78
79   public Long getClipLength() {
80     return clip.getMicrosecondLength();
81   }
82 }
```

Kodenya sedikit panjang, yang perlu kita perhatikan adalah fitur yang ditawarkan dari kelas `MyPlayer` ini, perhatikan pada baris ke-14 bahwa konstruktor membutuhkan *path* dari berkas audio yang didukung oleh Java. Di dalam konstruktor kita menyiapkan segala sesuatunya, mulai dari *input stream* yang digunakan untuk membaca dari sebuah berkas, kemudian menyiapkan instan kelas `Clip` untuk memutar audio, membuka berkas, dan melakukan konfigurasi untuk selalu memutar berkas dengan parameter `Clip.LOOP_CONTINUOUSLY`.

Pada blok baris ke-22 sampai ke-25 adalah fitur yang ditawarkan untuk mulai memutar sebuah berkas audio. Di dalamnya kita memerintahkan instan kelas `Clip`

untuk mulai memutarkan berkas audio, kemudian memberikan sebuah status `play` ke pengguna.

Pada blok baris ke-27 sampai ke-35 adalah fitur untuk menghentikan sejenak pemutaran berkas audio yang sedang berjalan. Sedangkan pada baris ke-37 sampai ke-46 adalah untuk melanjutkan pemutarannya setelah melakukan penghentian sementara pada blok baris sebelumnya.

Pada blok baris ke-48 sampai ke-55 adalah melakukan pemutaran ulang dari posisi awal untuk berkas audio yang sedang dimuat.

Pada blok baris ke-57 sampai ke-61 adalah untuk menghentikan *player* memutar sebuah berkas audio.

Method pada blok baris ke-63 sampai ke-72 adalah untuk melompat ke detik tertentu dari bagian berkas audio. Sedangkan blok baris ke-74 sampai ke-77 adalah untuk memuat ulang berkas audio yang sudah ditentukan.

Kemudian kita membuat kelas `Aplikasi` yang akan menjadi panduan pengguna dalam melakukan operasi pemutaran berkas audio. Berikut adalah kodenya :

```
1  import java.io.*;
2  import java.util.Scanner;
3  import javax.sound.sampled.*;
4  import audio.MyPlayer;
5
6  public class Aplikasi {
7    private static MyPlayer audioPlayer;
8
9    public static void main(String args[]) {
10     try {
11       String filePath = "./audio/sample.au";
12       audioPlayer = new MyPlayer(filePath);
13       audioPlayer.play();
14       Scanner sc = new Scanner(System.in);
15
```

```
16      while(true) {
17      System.out.println("\n=== Menu");
18        System.out.println("1. pause");
19        System.out.println("2. resume");
20        System.out.println("3. restart");
21        System.out.println("4. stop");
22        System.out.println("5. Jump to specific time");
23        System.out.print("=> ");
24        int c = sc.nextInt();
25        gotoChoice(c);
26        if(c == 4) break;
27      }
28      sc.close();
29    } catch(Exception e) {
30      System.out.println("Kesalahan memutar berkas audio");
31      e.printStackTrace();
32    }
33  }
34
35  private static void gotoChoice(int c) throws
     UnsupportedAudioFileException, IOException,
     LineUnavailableException {
36    switch(c) {
37      case 1:
38        audioPlayer.pause();
39        break;
40      case 2:
41        audioPlayer.resumeAudio();
42        break;
43      case 3:
44        audioPlayer.restart();
45        break;
46      case 4:
```

```
47          audioPlayer.stop();
48          break;
49        case 5:
50          System.out.println("Masukkan waktu (" + 0 +
51            ", " + audioPlayer.getClipLength() + ")");
52          Scanner sc = new Scanner(System.in);
53          long c1 = sc.nextLong();
54          audioPlayer.jump(c1);
55          break;
56      }
57    }
58 }
```

Pada baris ke-11 kita menyiapkan berkas audio yang akan dijalankan, kemudian membentuk instan dari kelas `MyPlayer` kemudian menjalankan berkas audio tersebut dengan perintah `play()`.

Pada baris ke-14, kita menyiapkan `Scanner` untuk menerima masukkan dari pengguna. Kemudian pada blok baris ke-16 sampai ke-26 adalah untuk menampilkan menu aplikasi.

Hasil masukkan dari pengguna berupa angka akan dikirimkan ke *method* `gotoChoice` untuk kemudian diproses, apakah berkas akan dimainkan, dihentikan sementara, dijalankan kembali dari posisi penghentian sementara, menghentikan pemutaran audio, atau melompat pada waktu tertentu dari berkas audio yang diputar.

Semua operasi tersebut membutuhkan instan kelas `MyPlayer` yang sudah dibangun dan dieksekusi setiap fiturnya pada *method* `gotoChoice()`.

11.3.3 Animasi

Animasi sebetulnya adalah gambar yang berganti secara cepat, pada praktek kali ini kita akan menggunakan sebuah gambar yang nantinya akan dibuat seperti ber-

gerak. Kelas yang pertama adalah kelas `Board` yang merupakan pewarisan dari kelas `JPanel`, yang nantinya akan kita masukkan dalam sebuah *window* aplikasi, kemudian untuk frekuensi atau jeda waktu tertentu, objek gambar yang dimasukkan akan bergerak. Berikut adalah isi kode dari kelas `Board` :

```
1  package anim;
2
3  import java.awt.Color;
4  import java.awt.Dimension;
5  import java.awt.Graphics;
6  import java.awt.Image;
7  import java.awt.Toolkit;
8  import java.awt.event.ActionEvent;
9  import java.awt.event.ActionListener;
10 import javax.swing.ImageIcon;
11 import javax.swing.JPanel;
12 import javax.swing.Timer;
13
14 public class Board extends JPanel implements ActionListener {
15
16   private final int B_WIDTH = 350;
17   private final int B_HEIGHT = 350;
18   private final int INITIAL_X = -40;
19   private final int INITIAL_Y = -40;
20   private final int DELAY = 25;
21
22   private Image star;
23   private Timer timer;
24   private int x, y;
25
26   public Board() {
27     initBoard();
28   }
```

```
29
30  private void initBoard() {
31    setBackground(Color.BLACK);
32    setPreferredSize(new Dimension(B_WIDTH, B_HEIGHT));
33
34    loadImage();
35
36    x = INITIAL_X;
37    y = INITIAL_Y;
38
39    timer = new Timer(DELAY, this);
40    timer.start();
41  }
42
43  private void loadImage() {
44    ImageIcon ii = new ImageIcon("anim/star.png");
45    star = ii.getImage();
46  }
47
48  @Override
49  public void paintComponent(Graphics g) {
50    super.paintComponent(g);
51    drawStar(g);
52  }
53
54  private void drawStar(Graphics g) {
55    g.drawImage(star, x, y, this);
56    Toolkit.getDefaultToolkit().sync();
57  }
58
59  @Override
60  public void actionPerformed(ActionEvent e) {
61    x++; y++;
```

```
62      if(y > B_HEIGHT) {
63        y = INITIAL_Y;
64        x = INITIAL_X;
65      }
66      repaint();
67    }
68
69  }
```

Perhatikan pada konstruktor di baris ke-26, di dalamnya langsung memanggil sebuah *method* `initBoard()` yang fungsinya untuk mempersiapkan objek dari `Board` apabila dibentuk sebuah instan.

Di dalam *method* `initBoard()`, kita menyiapkan warna latar belakang berupa warna hitam, dan membuat ukuran jendelanya sebesar `B_WIDTH` x `B_HEIGHT`. Kemudian memuat sebuah gambar dengan memanggil *method* `loadImage()`.

Dalam *method* `initBoard()` pun kita menyiapkan `Timer` dengan 2 (dua) parameter, parameter pertama adalah `DELAY`, dimana nantinya akan ada jeda waktu selama `DELAY` dalam satuan mikro-detik, kemudian parameter kedua adalah `this`, yang menunjuk pada `ActionListener`, karena nantinya `timer` akan melakukan iterasi terus menerus pada *method* `actionPerformed()` milik kelas `ActionListener` dengan jeda `DELAY`.

Method `actionPerformed()` telah kita *override* oleh kelas `Board`, sehingga nantinya `timer` akan dilakukan iterasi atau perulangan terus-menerus terhadap *method* `actionPerformed()` ini.

Dalam *method* `actionPerformed()`, kita mengubah posisi `x` dan `y` dan kemudian memanggil *method* `repaint()`, dimana *method* `repaint()` ini sebetulnya akan memanggil *method* `paintComponent()`.

Di dalam *method* `paintComponent()` kita membentuk ulang tampilan `Board` dengan memanggil *method* `paintComponent` milik `JPanel` dengan pernyataan

`super.paintComponent()` pada baris ke-50, kemudian menggambar ulang `star` dengan memanggil *method* `drawStar()` pada baris ke-51.

Hasil akhirnya pada saat program kita jalankan, akan ada gambar bintang yang bergerak dari pojok kiri atas ke pojok kiri bawah.

11.4 Kesimpulan

Bahwa implementasi penggunaan berkas gambar, audio, dan pembuatan animasi dapat dilakukan pada bahasa pemrograman Java hanya dengan paket yang sudah terintegrasi dalam JDK, walau memang kondisi format berkas yang didukung masih terbatas.

11.5 Tugas

Pada praktek pembuatan animasi, buatlah agar gambar bintang bergerak dari kiri ke kanan.

Bibliografi

[1] -. *OOPs Concept in Java*. https://beginnersbook.com/2013/04/oops-concepts/ (diakses pada tanggal 10 Maret 2019).

[2] Lemay, Laura. Perkins, Charles L. Morrison, Michael. *Teach Yourself Java in 21 Days*. http://www.dmc.fmph.uniba.sk/public_html/doc/Java/ (diakses pada tanggal 17 Maret 2019).