

MODUL PRAKTIKUM

PEMROGRAMAN BERORIENTASI OBJEK

S1 INFORMATIKA

Class

Java

{ }

Official Line
Informatics Lab



Published by school of computing

Lembar Pengesahan

Saya yang bertanda tangan di bawah ini:

Nama : Dr. Dody Qory Utama, S.T., M.T.
NIK : 14870074
Koordinator Mata Kuliah : Pemrograman Berbasis Objek
Prodi : S1 Informatika

Menerangkan dengan sesungguhnya bahwa modul ini digunakan untuk pelaksanaan praktikum di Semester Ganjil Tahun Ajaran 2021/2022 di Laboratorium Informatika Fakultas Informatika Universitas Telkom.

Bandung, September 2021



Fakultas Informatika
School of Computing
Telkom University



Mengetahui,
Kaprodi S1 Informatika

Pemrograman Berbasis Objek

Dr. Dody Qory Utama, S.T., M.T.

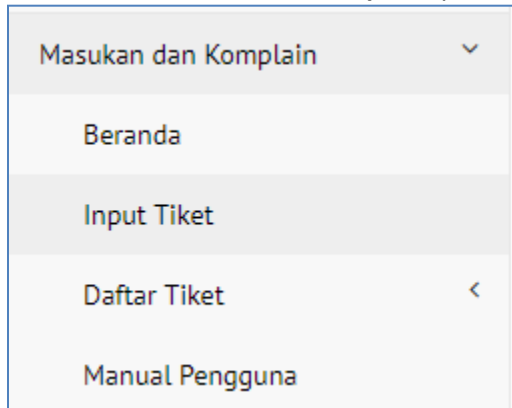
Dr. Erwin Budi Setiawan, S.Si., M.T.

Peraturan Praktikum Laboratorium Informatika 2020/2021

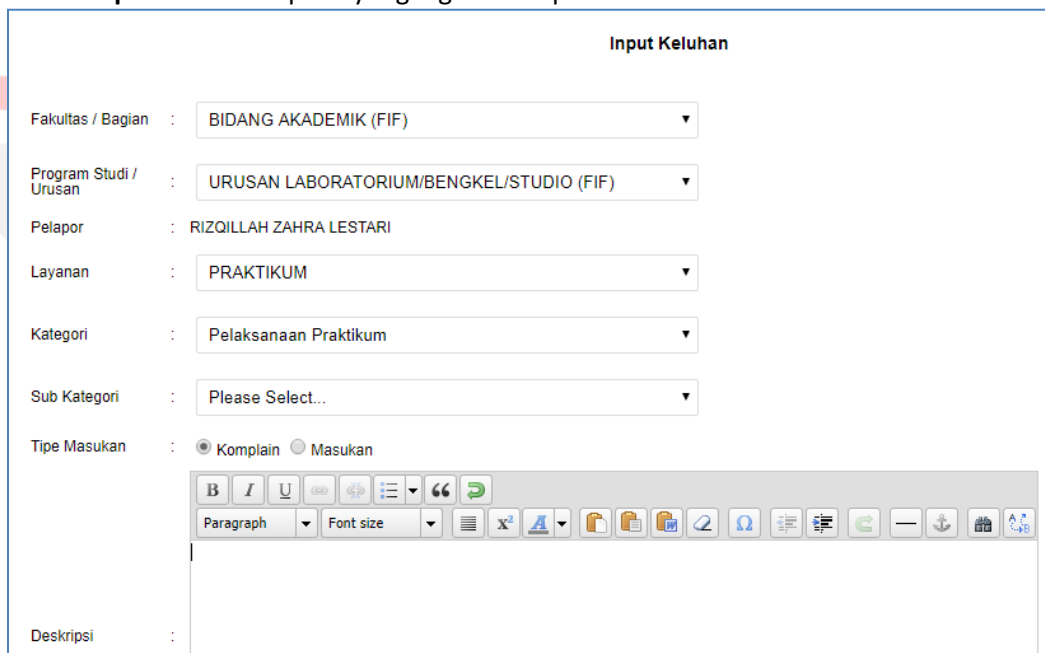
1. Praktikum diampu oleh dosen kelas dan dibantu oleh asisten laboratorium dan asisten praktikum.
2. Praktikum dilaksanakan di Gedung Kultubai Selatan (IFLAB 1 s/d IFLAB 5) & Kultubai Utara (IFLAB 6 s/d IFLAB 7) sesuai jadwal yang ditentukan.
3. Praktikan wajib membawa modul praktikum, kartu tanda mahasiswa, dan alat tulis.
4. Praktikan wajib melakukan *tapping* menggunakan kartu tanda mahasiswa diawal praktikum.
5. Praktikan wajib mengisi daftar hadir *rooster* praktikum dengan bolpoin bertinta hitam.
6. Durasi kegiatan praktikum S1 = 2 jam (100 menit).
7. Jumlah pertemuan praktikum:
 - a. 10 kali di lab (praktikum rutin atau pertemuan kedua sampai pertemuan kesebelas).
 - b. 3 kali di luar lab (terkait Tugas Besar dan UAS).
 - c. 1 kali Running Modul (pertemuan pertama).
8. Presensi praktikum termasuk presensi matakuliah.
9. Praktikan yang datang terlambat tidak mendapat tambahan waktu.
10. Saat praktikum berlangsung, asisten praktikum dan praktikan:
 - a. Wajib menggunakan seragam sesuai aturan institusi.
 - b. Wajib mematikan atau mengondisikan semua alat komunikasi.
 - c. Dilarang membuka aplikasi yang tidak berhubungan dengan praktikum yang berlangsung.
 - d. Dilarang mengubah pengaturan *software* maupun *hardware* komputer tanpa ijin.
 - e. Dilarang membawa makanan maupun minuman di ruang praktikum.
 - f. Dilarang memberikan jawaban ke praktikan lain.
 - g. Dilarang menyebarkan soal praktikum.
 - h. Dilarang membuang sampah di ruangan praktikum.
 - i. Dilarang mencoret, mengotori, atau merusak fasilitas laboratorium.
 - j. Wajib meletakkan alas kaki dengan rapi pada tempat yang telah disediakan.
11. Setiap praktikan dapat mengikuti praktikum susulan maksimal dua modul untuk satu mata kuliah praktikum.
 - a. Praktikan yang dapat mengikuti praktikum susulan hanyalah praktikan yang memenuhi syarat sesuai ketentuan institusi, yaitu: sakit (dibuktikan dengan surat keterangan medis), tugas dari institusi (dibuktikan dengan surat dinas atau dispensasi dari institusi), atau mendapat musibah atau kedukaan (menunjukkan surat keterangan dari orangtua atau wali mahasiswa.)
 - b. Persyaratan untuk praktikum susulan diserahkan sesegera mungkin kepada asisten laboratorium untuk keperluan administrasi.
 - c. Praktikan yang diijinkan menjadi peserta praktikum susulan ditetapkan oleh Asman Lab dan Bengkel Informatika dan tidak dapat diganggu gugat.
12. Pelanggaran terhadap peraturan praktikum akan ditindak secara tegas secara berjenjang di lingkup Kelas, Laboratorium, Fakultas, hingga Universitas.
13. Website IFLAB : <https://informatics.labs.telkomuniversity.ac.id/>
Asman IFLAB (Sidik Prabowo) : 081322744212, pakwowo@telkomuniversity.ac.id
Laboran IFLAB (Oku Dewi) : 085294057905, laboratorium.informatika@gmail.com

Tata Cara Komplain Praktikum IFLAB Melalui IGRACIAS

1. Login Igracias.
2. Pilih Menu **Masukan dan Komplain**, pilih **Input Tiket**.



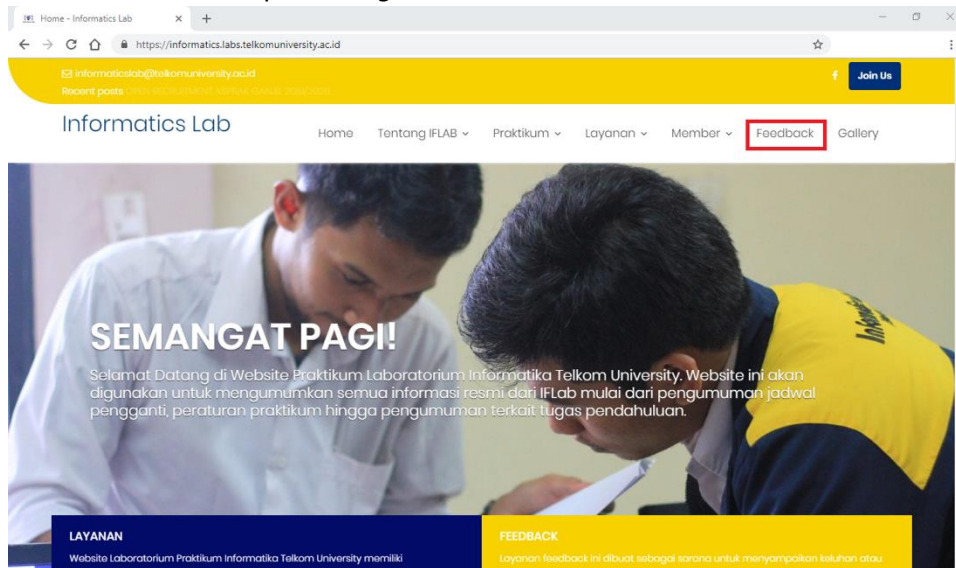
3. Pilih Fakultas/Bagian: **Bidang Akademik (FIF)**.
4. Pilih Program Studi/Urusan: **Urusan Laboratorium/Bengkel/Studio (FIF)**.
5. Pilih Layanan: **Praktikum**.
6. Pilih Kategori: **Pelaksanaan Praktikum**, lalu pilih **Sub Kategori**.
7. Isi **Deskripsi** sesuai komplain yang ingin disampaikan.

A screenshot of the 'Input Keluhan' form. The form contains several dropdown menus and a text input field. The fields are: 'Fakultas / Bagian' (set to 'BIDANG AKADEMIK (FIF)'), 'Program Studi / Urusan' (set to 'URUSAN LABORATORIUM/BENGKEL/STUDIO (FIF)'), 'Pelapor' (set to 'RIZQILLAH ZAHRA LESTARI'), 'Layanan' (set to 'PRAKTIKUM'), 'Kategori' (set to 'Pelaksanaan Praktikum'), 'Sub Kategori' (set to 'Please Select...'), and 'Tipe Masukan' (with radio buttons for 'Komplain' and 'Masukan', where 'Komplain' is selected). Below these fields is a rich text editor with a toolbar containing various icons for text formatting and a large text area for the 'Deskripsi'.

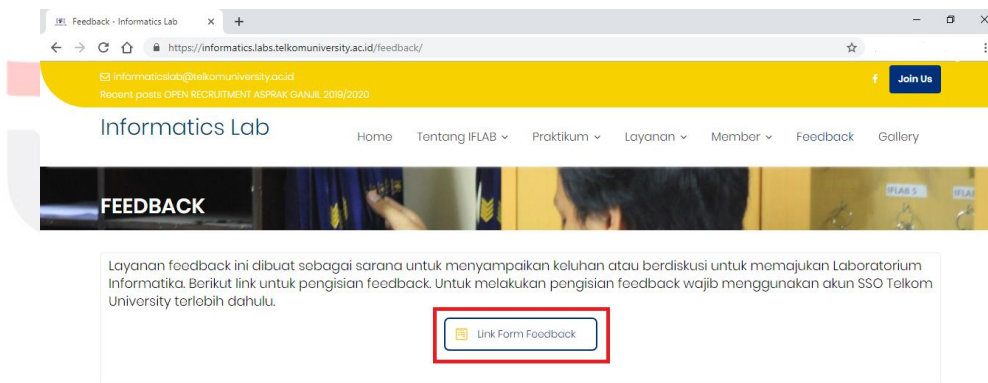
8. Lampirkan *file* jika perlu. Lalu klik Kirim.

Tata Cara Komplain Praktikum IFLAB Melalui Website

1. Buka website <https://informatics.labs.telkomuniversity.ac.id/> melalui browser.
2. Pilih menu **Feedback** pada *navigation bar website*.



3. Pilih tombol **Link Form Feedback**.



4. Lakukan *login* menggunakan akun **SSO Telkom University** untuk mengakses *form feedback*.
5. Isi *form* sesuai dengan *feedback* yang ingin diberikan.

Daftar Isi

Lembar Pengesahan.....	i
Peraturan Praktikum Laboratorium Informatika 2020/2021	ii
Tata Cara Komplain Praktikum IFLAB Melalui IGRACIAS	iii
Tata Cara Komplain Praktikum IFLAB Melalui <i>Website</i>	iv
Daftar Isi.....	v
Daftar Gambar	ix
Modul 1 Running Modul.....	1
1.1 Pengenalan Java.....	1
1.1.1 Sejarah Singkat Java	1
1.1.2 Karakteristik Java	1
1.1.3 Garbage Collection.....	2
1.2 Instalasi Netbeans.....	2
1.2.1 Instalasi JDK (Java Development Kit)	2
1.2.2 Netbeans.....	3
Modul 2 Intro Pemrograman Berorientasi Objek.....	5
2.1 Tipe Data dan Variabel.....	5
2.2 Konstanta	6
2.3 Variabel Array	6
2.3.1 Deklarasi Variabel Array.....	6
2.3.2 Mengakses Variabel Array	7
2.3.3 Array 2 Dimensi.....	7
2.3.4 Array Bertipe Objek.....	8
2.4 Operator, Pernyataan Kondisional, dan Perulangan	9
2.4.1 Operator.....	9
2.4.2 Pernyataan Kondisional	11
2.4.3 Perulangan	13
2.5 Compile, Run, dan Jar File	14
2.5.1 Compile	14
2.5.2 Run	14
2.5.3 Jar File	14
Modul 3 Konsep Dasar Pemrograman Berorientasi Objek	16
3.1 Pendahuluan	16
3.1.1 Abstraksi.....	16
3.1.2 Enkapsulasi.....	16
3.1.3 Pewarisan (Inheritance)	17
3.1.4 Polymorphisms.....	17

3.1.5	Message (Komunikasi Antar Objek)	18
3.2	Kelas	18
3.3	Object	18
3.4	Field	19
3.5	Method	19
3.6	Keyword “this”	21
Modul 4	Encapsulation	24
4.1	Java Modifier	24
4.2	Constructor	26
4.3	Package	28
Modul 5	Relasi Antar Kelas	30
5.1	Diagram Kelas	30
5.2	Hubungan Antar Kelas	31
5.2.1	Asosiasi	31
5.2.2	Agregasi	33
5.2.3	Komposisi	34
Modul 6	Inheritance	35
6.1	Pendahuluan	35
6.2	Inheritance	35
Keyword Super		36
6.3	Overloading Method	37
6.4	Overriding Method	38
Modul 7	Abstract dan Interface	40
7.1	Kelas Abstrak	40
7.2	Interface	42
Modul 8	Polymorphism	47
8.1	Pendahuluan	47
8.2	Referensi Variabel Casting	47
Modul 9	Inner Class, Collection, dan Generics	50
9.1	Inner Class	50
9.2	Static	52
9.2.1	Static Variabel	52
9.2.2	Static <i>Method</i>	53
9.3	Collection	53
9.3.1	Melakukan Sorting pada Collection	55
9.3.2	Melakukan Filtering pada Collection	57
9.4	Generics	57

Modul 10	Exception.....	59
10.1	Exception.....	59
10.2	Eksepsi yang tidak dicek.....	59
10.3	Eksepsi yang dicek.....	60
10.4	Penggunaan Blok Try Catch.....	60
10.5	Penggunaan Throws.....	61
10.6	Pemakaian Finally	61
Modul 11	Graphical User Interface (GUI).....	63
11.1	AWT and Swing	63
11.2	AWT components.....	63
11.3	Swing components.....	63
11.4	Komponen utama dalam GUI.....	63
11.5	The Basic User Interface Components with Swing	64
11.6	Top Level Container	64
11.7	Pengaturan Layout.....	65
11.7.1	Pengaturan dengan BorderLayout.....	65
11.7.2	Pengaturan dengan BoxLayout	66
11.7.3	Pengaturan dengan CardLayout	67
11.7.4	Pengaturan dengan FlowLayout	68
11.7.5	Pengaturan dengan GridLayout	69
11.8	Event Handling	69
11.8.1	Button	70
11.8.2	ComboBox.....	71
11.8.3	CheckBox.....	73
11.8.4	RadioButton	74
11.8.5	Table.....	75
Modul 12	Input Output dan File.....	78
12.1	Input Output	78
12.1.1	Java I/O.....	78
12.1.2	Byte Stream.....	78
12.2	Operasi File	80
12.2.1	java.io.File	80
12.2.2	java.io.FileOutputStream	80
12.2.3	java.io.FileInputStream	81
12.3	Object Serialization	81
12.3.1	Interface java.io.Serializable	82
12.3.2	Class java.io.ObjectOutputStream	82

12.3.3	Class java.io.ObjectInputStream	82
Modul 13	JDBC (Java Database Connectivity)	84
Modul 14	Tugas Besar	87
14.1	Implementasi JDBC dengan GUI	87
Daftar Pustaka	90



Fakultas Informatika
 School of Computing
 Telkom University



Daftar Gambar

Gambar 1-1 Tampilan awal instalasi Java (JDK 1.7).....	2
Gambar 1-2 Pemilihan tempat penginstalan Java.	3
Gambar 1-3 Tampilan Dialog <i>New Project</i>	3
Gambar 1-4 Tampilan Run Main Project.....	4
Gambar 1-5 Tampilan hasil <i>compile</i> dan <i>run</i> program <i>hello world</i>	4
Gambar 4-1 Struktur <i>file</i> program harga token dan harga pulsa.....	28
Gambar 5-1 Diagram Kelas.	30
Gambar 5-2 Hubungan Asosiasi.	30
Gambar 5-3 Hubungan Agregasi.	30
Gambar 5-4 Hubungan Komposisi.	31
Gambar 5-5 Hubungan Generalisasi.	31
Gambar 5-6 Contoh hubungan asosiasi.	31
Gambar 6-1 Ilustrasi pada kelas manusia.	35
Gambar 6-2 Contoh kelas dengan <i>keyword super</i>	36
Gambar 7-1 Contoh kelas abstrak dalam UML.	40
Gambar 7-2 Contoh kelas <i>interface</i> dalam UML.....	43
Gambar 9-1 Hierarki dari <i>Collection</i>	54
Gambar 10-1 Hierarki dari tipe-tipe eksepsi.....	59
Gambar 11-1 Komponen <i>Swing</i>	63
Gambar 11-2 BorderLayout.	65
Gambar 11-3 BorderLayout.	66
Gambar 11-4 CardLayout.....	67
Gambar 11-5 FlowLayout.....	68
Gambar 11-6 GridLayout.....	69
Gambar 11-7 Contoh Penggunaan Button.....	70
Gambar 11-8 Contoh Penggunaan ComboBox.	72
Gambar 11-9 Contoh Penggunaan CheckBox.	73
Gambar 11-10 Contoh Penggunaan RadioButton.	74
Gambar 11-11 Contoh Penggunaan Table.....	75
Gambar 12-1 Alur membaca Stream.	78
Gambar 12-2 Alur menulis Stream.	78
Gambar 12-3 Hirarki kelas dari class <i>InputStream</i>	79
Gambar 12-4 Hirarki kelas dari class <i>OutputStream</i>	79
Gambar 14-1 Contoh implementasi sederhana dari JDBC dan GUI.....	89

Modul 1 Running Modul

Tujuan Praktikum

1. Mengetahui dan mengenal bahasa pemrograman java
2. Mengetahui cara instalasi dan konfigurasi java

1.1 Pengenalan Java

1.1.1 Sejarah Singkat Java



Java dibuat dan diperkenalkan pertama kali oleh sebuah tim Sun Microsystem yang dipimpin oleh **Patrick Naughton** dan **James Gosling** pada tahun 1991 dengan nama kode **Oak**. Tahun 1995 Sun mengubah nama Oak tersebut menjadi **Java**.

Teknologi java diadopsi oleh Netscape tahun 1996. JDK 1.1 diluncurkan tahun 1996, kemudian JDK 1.2, berikutnya J2EE (Java 2 Enterprise Edition) yang berbasis J2SE yaitu servlet, EJB dan JSP. Dan yang terakhir adalah J2ME (Java 2 Micro Edition) yang diadopsi oleh Nokia, Siemens, Motorola, Samsung, dan Sony Ericson



Tahukah kamu?

Ide pertama kali kenapa Java dibuat adalah karena adanya motivasi untuk membuat sebuah bahasa pemrograman yang bersifat *portable* dan *platform independent* (tidak tergantung mesin dan sistem operasi) yang dapat digunakan untuk membuat piranti lunak yang dapat ditanamkan (*embedded*) pada berbagai macam peralatan *electronic consumer* biasa, seperti *microwave*, *remote control*, telepon, *card reader* dan sebagainya.

1.1.2 Karakteristik Java

Karakteristik Java adalah sebagai berikut :

- 1) Berorientasi Objek, Java telah menerapkan konsep pemrograman berorientasi objek yang modern dalam implementasinya.
- 2) *Robust*, java mendorong pemrograman yang bebas dari kesalahan dengan bersifat *strongly typed* dan memiliki *run-time checking*
- 3) *Portable*, Program Java dapat dieksekusi di platform manapun selama tersedia Java Virtual Machine untuk *platform* tersebut.
- 4) *Multithreading*, Java mendukung penggunaan *multithreading* yang telah terintegrasi langsung dalam bahasa Java
- 5) Dinamis, Program Java dapat melakukan suatu tindakan yang ditentukan pada saat eksekusi program dan bukan pada saat kompilasi
- 6) Sederhana, Java menggunakan bahasa yang sederhana dan mudah dipelajari.
- 7) Terdistribusi, java didesain untuk berjalan pada lingkungan yang terdistribusi seperti halnya internet.
- 8) Aman, Java memiliki arsitektur yang kokoh dan pemrograman yang aman.

1.1.3 Garbage Collection

Teknik yang digunakan Java untuk penanganan objek yang telah tidak diperlukan untuk dimusnahkan (dikembalikan ke *pool memory*) disebut garbage collection. *Java Interpreter* melakukan pemeriksaan untuk mengetahui apakah objek di memori masih diacu oleh program. *Java Interpreter* akan mengetahui objek yang telah tidak dipakai oleh program. Ketika *Java interpreter* mengetahui objek itu, maka *Java interpreter* akan memusnahkan secara aman.

Java garbage collector berjalan sebagai thread berprioritas rendah dan melakukan kerja saat tidak ada kerja lain di program. Umumnya, *Java garbage collector* bekerja saat idle pemakai menunggu masukan dari *keyboard* atau *mouse*. *Garbage collector* akan bekerja dengan prioritas tinggi saat interpreter kekurangan memori. Hal ini jarang terjadi karena *thread* berprioritas rendah telah melakukan tugas dengan baik secara background.

1.2 Instalasi Netbeans

Sebelum *install* Netbeans Kita perlu *install* JDK terlebih dahulu.

1.2.1 Instalasi JDK (Java Development Kit)

Pada Windows, *file* instalasi berupa **self-installing file**, yaitu *file exe* yang bila dijalankan akan mengekstrak dirinya untuk dikopikan ke direktori. Prosedur instalasi akan menawarkan *default* untuk direktori instalasi seperti jdk 1.4, jdk 1.5, jdk 1.6 atau yang terbaru jdk 1.7. Kita sebaiknya tidak mengubahnya karena Kita dapat mengkoleksi beragam versi JDK, secara *default*.



Gambar 1-1 Tampilan awal instalasi Java (JDK 1.7).



Gambar 1-2 Pemilihan tempat penginstalan Java.

Selanjutnya klik “Next” dan tinggal mengikuti semua alur instalasi sampai proses instalasi selesai.

1.2.2 Netbeans

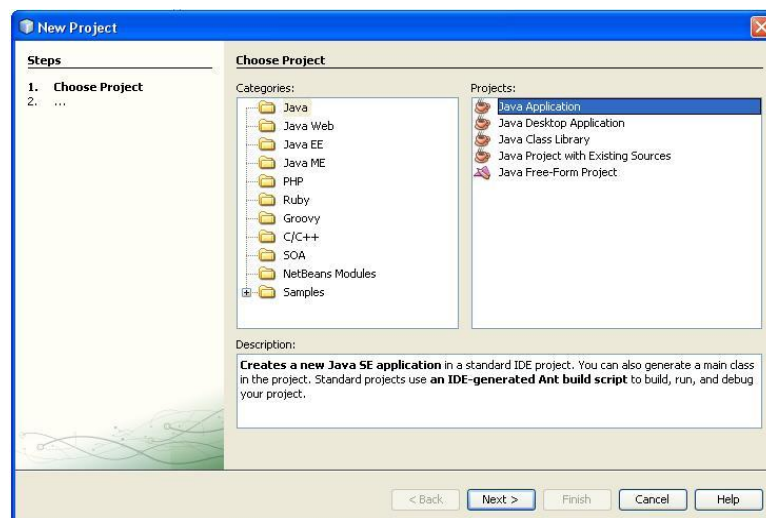
Setelah instalasi JDK selesai, tahap selanjutnya adalah melakukan instalasi Netbeans terbaru yang dapat diunduh di <http://www.netbeans.org>. Proses instalasi dapat dilakukan sama seperti instalasi *software* pada umumnya yang hanya mengikuti alur instalasi hingga proses selesai.

Di dalam netbeans, semua perancangan dan pemrograman dilakukan di dalam kerangka sebuah proyek. Proyek netbeans merupakan sekumpulan *file* yang dikelompokkan di dalam suatu kesatuan.

1) Membuat proyek

Sebelum membuat program Java dengan menggunakan Netbeans, langkah pertama adalah membuat proyek terlebih dahulu.

- a. Jalankan *menu File | Project* (Ctrl+Shift+N) untuk membuka suatu dialog *New Project*.



Gambar 1-3 Tampilan Dialog *New Project*.

- b. Dalam dialog, pilih *Categories : General* dan *Projects : Java Application*. Lalu klik tombol *next*.
- c. Pilih dahulu lokasi *project* dengan menekan tombol *browse*. Akan tampil direktori dan Kita dapat memilih di direktori mana *project* disimpan. Setelah memilih, klik *open*.
- d. Kembali ke dialog *new project*, isikan *project name*. Misalnya disini *project name : HelloWorld*. Perhatikan bahwa *project folder* secara otomatis akan diset sesuai dengan *project name*.
- e. Centang pada *Set as Main Project* dan pada *Create Main Class*. Terakhir klik tombol *finish*.

Di dalam netbeans akan dibuka secara otomatis *file* utama Java bernama *main.java* yang berada pada *package helloworld*.

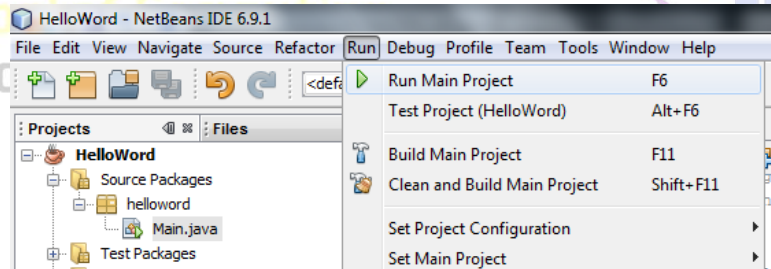
2) Memulai membuat program sederhana di Java

Setelah membuat proyek di Netbeans, selanjutnya membuat program sederhana. Di bawah ini merupakan contoh program yang akan menampilkan "*Hello Word*".

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello Word");
    }
}
```

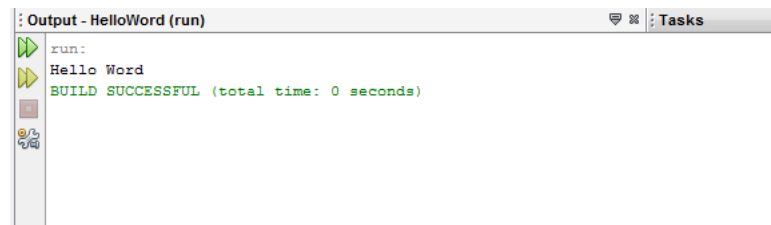
3) Meng-compile dan menjalankan program Java di Netbeans

Untuk meng-compile dan menjalankan klik *Run* → *Run Main Project*, seperti gambar dibawah ini:



Gambar 1-4 Tampilan Run Main Project.

Jika programnya sukses maka akan tampil tulisan *hello word* seperti di bawah ini :



Gambar 1-5 Tampilan hasil compile dan run program *hello world*.

Modul 2 Intro Pemrograman Berorientasi Objek

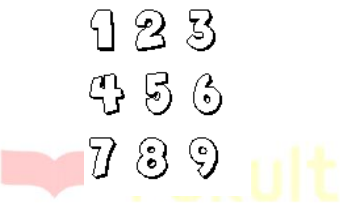
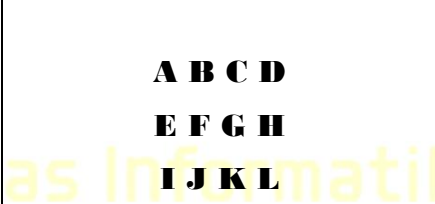

Tujuan Praktikum

1. Memahami tipe data, variabel, konstanta dan dapat mengimplementasikannya dalam pemrograman java.
2. Memahami dan mengimplementasikan operator, variabel array, pernyataan kondisional dan perulangan dalam pemrograman java.

2.1 Tipe Data dan Variabel

Hampir bisa dipastikan bahwa sebuah program selalu membutuhkan lokasi memori untuk menyimpan data yang sedang diproses. Kita tidak pernah tahu di lokasi memori mana komputer akan meletakkan data dari program Kita. Kenyataan ini menimbulkan kesulitan bagi Kita untuk mengambil data tersebut pada saat dibutuhkan. Maka dikembangkan konsep variabel. Setiap variabel memiliki jenis alokasi memori tersendiri, misalnya bilangan bulat, bilangan pecahan, karakter, dan sebagainya. Ini sering disebut dengan tipe data.

Secara umum ada tiga bentuk data:

 Numerik	 Karakter	 Logika
---	---	--

- 1) Numerik, data yang berbentuk angka atau bilangan. Data numerik bisa dibagi atas dua kategori :
 - a. Bilangan bulat (*integer*), yaitu bilangan yang tidak mengandung angka pecahan.
 - b. Bilangan pecahan (*float*), yaitu bilangan yang mengandung angka pecahan.
- 2) Karakter, data yang berbentuk karakter atau deretan karakter. Karakter bisa dibagi menjadi dua kategori :
 - a. Karakter tunggal.
 - b. Deretan karakter.
- 3) Logika, yaitu tipe data dengan nilai benar (*true*) atau salah (*false*).
Pada dasarnya ada dua macam tipe variabel data dalam bahasa Java, yakni:
 - a. Tipe primitif, meliputi : tipe boolean, tipe numerik (yang meliputi : Byte, short, int, long, char, float, double) dan tipe karakter (char).
 - b. Tipe referensi, meliputi tipe variabel data : tipe kelas, tipe array, tipe interface. Ada pula tipe variabel data khusus yang disebut null types, namun variabel dalam Java tidak akan memiliki tipe *null* ini.

Bentuk umum pendeklarasian variabel:

```
tipeData namaVariabel1 =[nilaiAwal];  
tipeData namaVariabel1 =[nilaiAwal], [namaVariabel2 = [nilaiAwal], ...];
```

Contoh untuk mendeklarasikan sebuah variabel :

```
int dataint;  
char chardata;  
float x = 12,67;
```


Penamaan sebuah variabel dalam Java harus memenuhi aturan sebagai berikut :

- 1) Harus terdiri atas sederetan karakter Unicode yang diawali oleh karakter huruf atau garis bawah. *Unicode* merupakan sistem pengkodean karakter yang dapat dibaca oleh berbagai bahasa manusia.
- 2) Tidak boleh berupa *keyword* (kata yang dicadangkan), *null*, atau *literal true/false*.
- 3) Harus unik dalam suatu *scope*.

Dalam gaya pemrograman dengan Java, biasanya mengikuti format nama variabel, yaitu:

- 1) Diawali dengan huruf kecil
- 2) Jika nama variabel lebih dari satu kata, kata ke-2, ke-3 dan seterusnya diawali dengan huruf kapital dan ditulis menyatu.

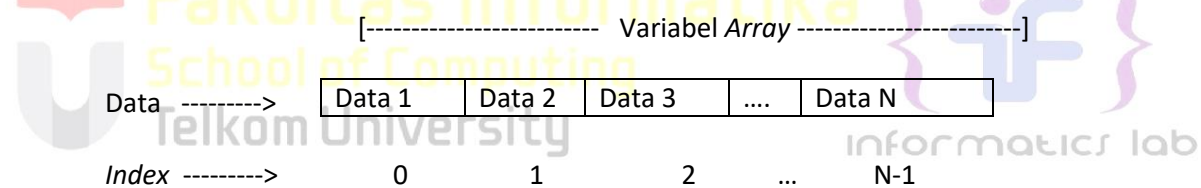
2.2 Konstanta

Variabel dalam Java bisa dijadikan konstanta, sehingga nilainya tidak akan dapat diubah-ubah dengan mendeklarasikannya sebagai variabel *final* seperti contoh :

```
final int x = 2;
```

2.3 Variabel Array

Untuk memudahkan pemahaman mengenai variabel *array*, digunakan ilustrasi kotak yang merepresentasikan setiap elemen variabel *array*. Sebuah variabel *array* bisa digambarkan berupa kotak sebagai berikut :



Index adalah sebuah angka yang menyatakan urutan sebuah elemen pada suatu variabel *array*. Karena seluruh kotak memiliki nama yang sama, maka untuk membedakannya diperlukan suatu cara yaitu dengan memberi nomor urut. Ibarat deretan rumah pada sebuah jalan, untuk membedakan antara rumah yang satu dengan rumah yang lain maka setiap rumah diberi nomor unik yang berbeda antara rumah satu dengan yang lainnya.

2.3.1 Deklarasi Variabel Array

Bentuk umum pendeklarasian variabel *array* di Java adalah :

```
tipeData[] namaVariabel = new tipeData [jumlahElemen];  
// atau  
tipeData namaVariabel[] = new tipeData [jumlahElemen];
```

Tipe data bisa berupa salah satu dari berbagai tipe data seperti *int*, *long*, *double* maupun nama kelas; baik kelas standar Java atau kelas buatan Kita sendiri. Materi kelas tidak akan dibahas dalam modul ini.

Cara pendeklarasian variabel *array* :

- 1) Mendeklarasikan variabel *array* tanpa menyebutkan berapa jumlah elemen yang diperlukan.

```
int[] variabelArray1;
```

Untuk deklarasi variabel *array* dengan cara ini, Kita harus menuliskan di salah satu baris program instruksi untuk memesan jumlah elemen untuk variabel tersebut. Sebelum terjadi pemesanan jumlah elemen, Kita tidak bisa menggunakan variabel *array* ini untuk menyimpan data.

```
variabelArray1 = new int[5];
```

- 2) Mendeklarasikan variabel *array* dengan menyebutkan jumlah elemen yang diperlukan

```
int[] variabelArray2 = new int[5];
```

Pada saat mendeklarasikan variabel *array* ini, maka Kita langsung memesan 5 elemen *array* yang akan Kita gunakan selanjutnya.

- 3) Mendeklarasikan variabel *array* secara otomatis

```
int[] variabelArray3 = {5, 3, 23, 99, 22};  
// atau  
int[] variabelArray3 = new int[]{1,23,45,4,3};
```

Variabel *array* ini memiliki 5 elemen, dan tiap elemen sudah terisi data. Berbeda dengan cara pendeklarasian yang pertama dan yang kedua.

2.3.2 Mengakses Variabel Array

Mengakses suatu variabel berarti mengisi data ke variabel tersebut atau mengambil data dari variabel tersebut. Mengakses variabel *array* berarti Kita mengakses elemen dari variabel *array* tersebut.

```
int tampung = var1[2];
```

Untuk mengisi nilai ke variabel *array*, sebutkan nomor indeks elemen yang akan diisi dengan nilai.

```
var1[5]=65
```

Untuk mengambil nilai dari variabel *array*, sebutkan nomor indeks elemen yang akan Kita ambil isinya.

```
var1[0] = var1[5] + var1[9];
```

Instruksi di atas akan menjumlahkan isi variabel *var1* elemen ke-6 dan ke-10 lalu menyimpan hasilnya ke elemen pertama.

2.3.3 Array 2 Dimensi

Bentuk umum pendeklarasian variabel *array* dua dimensi di Java adalah :

Pendeklarasian *array* dua dimensi bentuk **Rectangular**

```
tipeData[][] namaVariabel = new tipeData[jumlahBaris][jumlahKolom];  
// atau  
tipeData namaVariabel[][] = new tipeData[jumlahBaris][jumlahKolom];
```

Pendeklarasian *array* dua dimensi bentuk **Non-Rectangular**

```
tipeData [ ][ ] namaVariabel = new tipeData [jumlahBaris];  
namaVariabel [ ] = new tipeData[jumlahBaris];
```

Didalam *array* dua dimensi terdapat *array rectangular* dan *non-rectangular*. *Array* dua dimensi secara **Rectangular** artinya ukuran dimensi *array* semuanya sama yakni ukuran indeks pada baris dan kolom sama. Sedangkan **Non-Rectangular** artinya membuat *array* dengan ukuran indeks baris dan kolom yg tidak sama.

Cara pendeklarasian variabel *array* dua dimensi sama dengan cara pendeklarasian variabel *array* satu dimensi. Mengakses variabel *array* dua dimensi, mengisi variabel *array* dua dimensi,

mengambil variabel *array* dua dimensi juga sama dengan cara yang berlaku pada variabel *array* satu dimensi, yaitu dengan menyebutkan nomor indeks dari elemen. Perbedaanya hanya pada dua dimensi harus menyebutkan indeks baris dan indeks kolom.

Menghitung jumlah elemen pada *array* dua dimensi.

```
long[][] gede = new long[5][5]; /*deklarasi variabel array dua dimensi*/
gede.length;                  // akan melaporkan jumlah baris, sedangkan
gede[i].length;               // akan melaporkan jumlah kolom pada baris ke-i
```

Contoh array 2 dimensi *rectangular* :

```
public class Array2
{
    public static void main(String[] args)
    {
        double m[][];
        m = new double[4][4];
        m[0][0] = 1;
        m[1][1] = 1;
        m[2][2] = 1;
        m[3][3] = 1;
        System.out.println(m[0][0]+" "+m[0][1]+" "+m[0][2]+" "+m[0][3]);
        System.out.println(m[1][0]+" "+m[1][1]+" "+m[1][2]+" "+m[1][3]);
        System.out.println(m[2][0]+" "+m[2][1]+" "+m[2][2]+" "+m[2][3]);
        System.out.println(m[3][0]+" "+m[3][1]+" "+m[3][2]+" "+m[3][3]);
    }
}
```

Contoh array 2 dimensi *non-rectangular*:

```
public class Array2
{
    public static void main(String[] args)
    {
        int twoDim [][] = new int[2][];
        twoDim[0] = new int[2];
        twoDim[1] = new int[3];

        twoDim[0][0] = 1 ;
        twoDim[0][1] = 4 ;
        twoDim[1][0] = 1 ;
        twoDim[1][1] = 4 ;
        twoDim[1][2] = 4 ;
        System.out.println(twoDim[0][0]);
        System.out.println(twoDim[0][1]);
        System.out.println(twoDim[1][0]);
        System.out.println(twoDim[1][1]);
        System.out.println(twoDim[1][2]);
    }
}
```

2.3.4 Array Bertipe Objek

Array juga bisa digunakan untuk menyimpan beberapa objek yang berasal dari kelas yang sama, sebagai contoh objek manusia memiliki *method* info sehingga dengan menggunakan *array* bertipe objek, objek manusia dapat dibentuk menjadi *array*. Berikut contoh program *array* bertipe objek.

```
//file : Manusia.java
public class Manusia {

    private String nama;
    private int tinggi;

    void setInfo(String nama, int tinggi){
```

```

        this.nama = nama;
        this.tinggi = tinggi;
    }

    void info(){
        System.out.println(this.nama+ " Memiliki Tinggi " + this.tinggi +"cm");
    }
}

```

```

//file : Main.java
public class Main {
    public static void main(String[] args){

        manusia [] manusia = new manusia[4];
        for (int i = 0; i < manusia.length; i++) {
            manusia[i] = new manusia();
        }
        manusia[0].setInfo("Hermawan", 180);
        manusia[1].setInfo("Suciati", 160);
        manusia[2].setInfo("Boy", 170);
        manusia[3].setInfo("Neneng", 165);

        manusia[0].info();
        manusia[1].info();
        manusia[2].info();
        manusia[3].info();
    }
}

```

Output program:

```

Hermawan Memiliki Tinggi 180cm
Suciati Memiliki Tinggi 160cm
Boy Memiliki Tinggi 170cm
Neneng Memiliki Tinggi 165cm

```

2.4 Operator, Pernyataan Kondisional, dan Perulangan

2.4.1 Operator

Operator-operator yang digunakan pada java ada beberapa jenis diantaranya yaitu:

- 1) Operator Aritmatika
- 2) Operator Relasional
- 3) Operator Kondisional
- 4) Operator *Shift* dan *Bitwise*
- 5) Operator *Assignment*

Operator Aritmatika

Operator ini digunakan pada operasi-operasi **aritmatika** seperti penjumlahan, pengurangan, pembagian dll. Operator-operator yang digunakan yaitu :

Operator	Contoh	Keterangan
+	a + b	Menambahkan a dengan b
-	a – b	Mengurangkan a dengan b
*	a * b	Mengalikan a dengan b
/	a / b	Membagi a dengan b
%	a % b	Menghasilkan sisa pembagian antara a dengan b

++	a ++	Menaikkan nilai a sebesar 1
--	a --	Menurunkan nilai a sebesar 1
-	- a	Negasi dari a.

Operator Relasional

Untuk membandingkan 2 nilai (variabel) atau lebih digunakan operator Relasional, dimana operator ini akan mengembalikan atau menghasilkan nilai **True** atau **False**.

Jenis-jenisnya yaitu:

Operator	Contoh	Keterangan
>	a > b	True jika a lebih besar dari b
<	a < b	True jika a lebih kecil dari b
>=	a >= b	True jika a lebih besar atau sama dengan b
<=	a <= b	True jika a lebih kecil atau sama dengan b
==	a == b	True jika a sama dengan b
!=	a != b	True jika a tidak sama dengan b

Operator Kondisional

Operator ini menghasilkan nilai yang sama dengan operator relasional, hanya saja penggunaannya lebih pada **operasi-operasi boolean**. Jenisnya yaitu:

Operator	Contoh	Keterangan
&&	a && b	True jika a dan b, keduanya bernilai True
	a b	True jika a atau b bernilai True
!	! a	True jika a bernilai False

Operator Shift dan Bitwise

Dua operator ini digunakan untuk **memanipulasi nilai** dari bitnya, sehingga diperoleh nilai yang lain.

Operator Shift		
Operator	Penggunaan	Deskripsi
>>	a >> b	Menggeser bit a ke kanan sejauh b
<<	a << b	Menggeser bit a ke kiri sejauh b
>>>	a >>> b	Menggeser bit a ke kanan sejauh b
Operator Bitwise		
Operator	Penggunaan	Deskripsi
&	a & b	Bitwise AND
	a b	Bitwise OR
^	a ^ b	Bitwise XOR
~	~a	Bitwise Complement

Operator Assignment

Operator assignment dalam Java digunakan untuk **memberikan sebuah nilai ke sebuah variabel**. Operator assignment hanya berupa '=', namun selain itu dalam Java dikenal beberapa *shortcut assignment operator* yang penting, yang digambarkan dalam table berikut :

Operator	Contoh	Ekivalen dengan
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
&=	a &= b	a = a & b
=	a = b	a = a b

<code>^=</code>	<code>a ^= b</code>	<code>a = a ^ b</code>
<code><<=</code>	<code>a <<= b</code>	<code>a = a << b</code>
<code>>>=</code>	<code>a >>= b</code>	<code>a = a >> b</code>
<code>>>>=</code>	<code>a >>>= b</code>	<code>a = a >>> b</code>

2.4.2 Pernyataan Kondisional

Statement *if*

Statement *if* memungkinkan sebuah program **untuk dapat memilih beberapa operasi untuk dieksekusi, berdasarkan beberapa pilihan**. Dalam bentuknya yang paling sederhana, bentuk *if* mengandung sebuah pernyataan tunggal yang dieksekusi jika ekspresi bersyarat adalah benar.

Bentuk *if* yang paling sederhana yaitu :

```
if ( ekspresi_kondisional ) {
    statement 1;
    statement 2;
    .....
}
```

Statement *if-else*

Untuk melakukan beberapa operasi yang berbeda jika salah satu **ekspresi_kondisional** bernilai salah, maka digunakan statement *else*. Bentuk *if-else* memungkinkan kode Java memungkinkan dua alternatif operasi pemrosesan : satu **jika statement bersyarat adalah benar dan satu jika salah**.

- 1) Bentuk statement *if- else* dengan 2 pilihan operasi pemrosesan.

```
if ( ekspresi_kondisional ){
    statement 1;
    statement 2;
    .....
}else{
    statement 1;
    statement 2;
    .....
}
```

- 2) Bentuk statement *if- else* dengan beberapa pilihan operasi pemrosesan.

```
if ( ekspresi_kondisional_A ){
    statement 1;
    statement 2;
    .....
}else if( ekspresi_kondisional_B ){
    statement 1;
    statement 2;
    .....
}else{
    statement 1;
    statement 2;
    .....
}
```

Contoh :

```
class IfElse {
    public static void main(String args[]) {
        int month = 4;
        String season;
        if (month == 12 || month == 1 || month == 2) {
            season = "Dingin";
        } else if (month == 3 || month == 4 || month == 5) {
            season = "Semi";
        } else if (month == 6 || month == 7 || month == 8) {
            season = "Panas";
        } else if (month == 9 || month == 10 || month == 11) {
            season = "Gugur";
        } else {
            season = "";
        }
        System.out.println("Bulan April masuk musim " + season + ".");
    }
}
```

Statement Switch

Bentuk umum pernyataan *switch* adalah sebagai berikut :

```
switch ( expression ) {
    case value_1 :
        statement 1;
        statement 2;
        . . .
        break;
    case value_2 :
        statement 1;
        statement 2;
        . . .
        break;
[default:]
        statement 1;
        statement 2;
        . . .
        break;}
}
```

Contoh :

```
int hari = 7;
String hariString;
switch (hari) {
    case 1: hariString = "Senin";
        break;
    case 2: hariString = "Selasa";
        break;
    case 3: hariString = "Rabu";
        break;
    case 4: hariString = "Kamis";
        break;
    case 5: hariString = "Jumat";
        break;
    case 6: hariString = "Sabtu";
        break;
    case 7: hariString = "Minggu";
        break;
    default: hariString = "Invalid month";
        break;
}
System.out.println(hariString);
```


Kata-kunci *case* menandai posisi kode dimana eksekusi diarahkan. Sebuah konstanta integer atau konstanta karakter/string mengikutinya, atau ekspresi yang mengevaluasi konstanta integer atau konstanta karakter/string. Sedangkan kata default ekuivalensi dengan kata *else* pada statement *if*.

Ekspresi bersyarat

Kita menggunakan sebuah ekspresi bersyarat untuk menggantikan sebuah bentuk *if-else*. Sintaks adalah sebagai berikut :

```
exp1 ? exp2 : exp3
```

Dalam hal ini, jika *exp1* adalah benar, *exp2* dieksekusi. Jika *exp1* adalah salah, *exp3* dieksekusi.

Contoh:

```
int a = 4;
int b = 5;
int nilai = 3 > 2 ? a : b;
```

Outputnya :

```
nilai = 4
```

2.4.3 Perulangan

1) While

```
while( expression ){
    statement 1;
    statement 2;
    .....
}
```

Selama **ekspresi benar** *while* akan **dieksekusi**.

2) Do

```
do{
    statement 1;
    statement 2;
    .....
}while( expression );
```

Hasil dari *while* akan **dikembalikan** kepada **do**. *Syntax do-while loop* :

3) For

```
for ( initialization ; expression ; step ){
    statement 1;
    statement 2;
    ...
}
```

Pada java terdapat 2 statement yang biasanya digunakan pada setiap bentuk iterasi diatas. Statement tersebut yaitu :

- break*, dapat menghentikan perulangan walaupun kondisi untuk berhenti belum terpenuhi.
- continue*, dengan *statement* ini Kita bisa melewati operasi yang dilakukan dalam iterasi sesuai dengan kondisi tertentu.

Contoh Perulangan: Menampilkan angka 1-10

```
public class angka {
    public static void main(String[] args) {
        int i;
        for(i=1;i<=10;i++){
            System.out.println(Integer.toString(i));
        }
        i=1;
        while(i<=10) {
            System.out.println(Integer.toString(i));
            i++;
        }
        i=1;
        do {
            System.out.println(Integer.toString(i));
            i++;
        }
        while(i<=10)
    }
}
```

2.5 Compile, Run, dan Jar File

Perlu diketahui bahwa Netbeans merupakan IDE Java yang sudah *user friendly*, maka dari itu sebelum mengenal Netbeans perlu mengetahui fundamental dari *Compile*, *Run*, dan *Jar File*. Pada sub bab 1.2.2 program *HelloWorld* dapat dijalankan dengan satu kali klik tombol *Run*, namun pada dasarnya agar suatu program Java dalam dijalankan harus melewati tahap *compile* lalu *run* dan untuk mempermudah *running* program maka dibuatlah *Jar File* dari program yang dibuat.

2.5.1 Compile

Compile pada java berarti mengubah *source code* ke *byte codes* agar dapat dipahami oleh Java VM. Jika proses *compile* berhasil, maka akan terdapat sebuah berkas file berekstensi *class*. Java tidak membuat *file executeable (.exe)* melainkan *file .class* tersebut. Jika tidak berhasil, lakukan perbaikan *source code*, lalu coba *compile* kembali

Pada contoh program *HelloWorld*, jalankan dengan perintah sebagai berikut

```
➔ javac HelloWorld
```

2.5.2 Run

Setelah program berhasil di-*Compile* ke dalam *byte code* Java, program dapat dialankan pada Java VM. Untuk menjalankan program Java, jalankan perintah Java namafile dengan namafile adalah *file .class* yang akan di-*Run* (tanpa ekstensi *.class*).

Pada contoh program *HelloWorld*, jalankan dengan perintah sebagai berikut

```
➔ java HelloWorld
```

2.5.3 Jar File

Java Archive (JAR) merupakan *file archive* yang menyimpan *source* program Java agar dapat dijalankan secara *portable* dengan langsung membuka *file Jar* secara langsung, namun pada beberapa kasus *Output* yang hanya ditampilkan di konsol, *Jar File* perlu dijalankan di CMD/Terminal. Pada Netbeans pembuatan *Jar File* cukup dengan melakukan *Build Project* atau menekan tombol F11 dan hasilnya dapat di lihat di direktori penyimpanan *source project*, namun dengan menggunakan CMD/Terminal ada beberapa langkah yang perlu dilakukan.

Pada contoh program *HelloWorld*, untuk membuat *Jar File*, perlu dilakukan *Compile* terhadap *file* Java yang dibuat, selanjutnya perlu dilakukan pembuatan *manifest* untuk menentukan *main class* dengan perintah `echo Main-Class: namamainclass >manifest.txt`. Pada contoh program *HelloWorld*, pastikan directory di CMD/Terminal sudah berpindah ke directory tempat *source* program Java lalu jalankan dengan perintah pembuatan *manifest* sebagai berikut

```
➔ echo Main-Class: Main >manifest.txt
```

Setelah *manifest* dibuat, maka *Jar File* dapat dibuat dengan perintah `jar cvfm namafilejar.jar manifest.txt *.class`. Pada contoh program *HelloWorld*, jalankan perintah pembuatan *jar* dengan perintah sebagai berikut

```
➔ jar cvfm HelloWorld.jar manifest.txt *.class
```

Lalu jalankan *file Jar* menggunakan perintah sebagai berikut

```
➔ java -jar HelloWorld.jar
```

Maka *Output* yang ditampilkan akan sama dengan *Run* pada program Netbeans, yaitu tulisan "Hello World".

Modul 3 Konsep Dasar Pemrograman Berorientasi Objek

Tujuan Praktikum
<ol style="list-style-type: none">1. Mengerti konsep dasar <i>Object Oriented Programming</i>.2. Dapat membandingkan antara pemrograman prosedural dengan pemrograman berorientasi <i>Object</i>.3. Memahami Kelas, <i>Object</i>, <i>Method</i> dan <i>Constructor</i>, serta mengimplementasikannya dalam suatu program Java.

3.1 Pendahuluan

Sebelum membahas tentang kelas terlebih dahulu Kita perlu mengetahui konsep pemrograman berorientasi objek. Suatu sistem yang dibangun dengan metode berorientasi objek adalah sebuah sistem yang komponennya di-enskapsulasi menjadi kelompok data dan fungsi, yang dapat mewarisi atribut dan sifat dari komponen lainnya dan komponen-komponen tersebut saling berinteraksi satu sama lain.

Beberapa karakteristik utama dalam pemrograman berorientasi objek yaitu:

3.1.1 Abstraksi

Abstraksi pada dasarnya adalah **menemukan hal-hal yang esensial pada suatu objek** dan mengabaikan hal-hal yang sifatnya insidental. Kita menentukan apa ciri-ciri (atribut) yang dimiliki oleh objek tersebut serta apa saja yang bisa dilakukan (metode/perilaku) oleh objek tersebut.

Contohnya adalah abstraksi pada manusia.

Ciri-ciri (atribut) : punya tangan, berat, tinggi, dll. (kata benda, atau kata sifat).

Fungsi (perilaku) : makan, minum, berjalan, dll. (kata kerja).

3.1.2 Enkapsulasi

Pengkapsulan adalah **proses pemaketan data objek bersama *method-methodnya***. Manfaat utama pengkapsulan adalah penyembunyian rincian-rincian implementasi dari pemakai/objek lain yang tidak berhak. Enkapsulasi memprotek suatu proses dari kemungkinan interferensi atau penyalahgunaan dari luar sistem.

Fungsi dari enkapsulasi adalah:

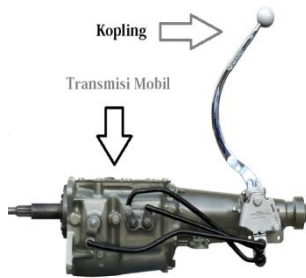
1) Penyembunyian data

Penyembunyian data (*data hiding*) mengacu perlindungan data internal objek. Objek tersebut disusun dari antarmuka *public Method* dan *private* data. Manfaat utama adalah bagian internal dapat berubah tanpa mempengaruhi bagian-bagian program yang lain.

2) Modularitas

Modularitas (*modularity*) berarti objek dapat dikelola secara **independen**. Karena kode sumber bagian internal objek dikelola secara terpisah dari antarmuka, maka Kita bebas melakukan modifikasi yang tidak menyebabkan masalah pada bagian-bagian lain. Manfaat ini mempermudah mendistribusikan objek-objek di sistem.

Contoh ilustrasi pada mobil :



Mobil memiliki sistem transmisi dan sistem ini menyembunyikan proses yang terjadi di dalamnya tentang bagaimana cara mobil tersebut bekerja, mulai dari bagaimana cara mobil mengatur percepatan dan apa yang dilakukan terhadap mesin untuk mendapat percepatan tersebut.

Kita sebagai pengguna hanya cukup memindah-mindahkan tongkat transmisi (kopling) untuk mendapatkan percepatan yang diinginkan.

Tongkat transmisi adalah satu-satunya *interface* yang digunakan dalam mengatur sistem transmisi mobil tersebut. Kita tidak dapat menggunakan pedal rem untuk mengakses sistem transmisi tersebut dan sebaliknya dengan mengubah-ubah sistem transmisi Kita tidak akan bisa menghidupkan radio mobil atau membuka pintu mobil. Konsep yang sama dapat pula Kita terapkan dalam pemrograman orientasi objek.

3.1.3 Pewarisan (Inheritance)

Pewarisan adalah **proses penciptaan kelas baru** (yang disebut subclass atau kelas turunan) **dengan mewarisi karakteristik** dari kelas yang telah ada (*superclass* atau kelas induk), ditambah karakteristik unik kelas baru itu. Karakteristik unik tersebut bisa merupakan perluasan atau spesialisasi dari *superclass*. Kelas turunan akan mewarisi anggota-anggota suatu kelas yang berupa data (atribut) dan fungsi (operasi) dan pada kelas turunan memungkinkan menambahkan data serta fungsi yang baru. Secara praktis berarti bahwa jika *superclass* telah mendefinisikan perilaku yang Kita perlukan, maka Kita tidak perlu mendefinisikan ulang perilaku itu, Kita cukup membuat kelas yang merupakan *subclass* dari *superclass* yang dimaksud.

3.1.4 Polymorphisms

Polymorphism berasal dari bahasa Yunani yang berarti **banyak bentuk**. Konsep ini memungkinkan digunakannya **antarmuka (*interface*) yang sama untuk memerintah suatu objek agar dapat melakukan aksi** atau tindakan yang mungkin secara prinsip sama tapi secara proses berbeda. Seringkali *Polymorphism* disebut dengan “satu interface banyak aksi”. Mekanisme *Polymorphism* dapat dilakukan dengan dengan beberapa cara, seperti *Overloading Method*, *Overloading Constructor*, maupun *Overriding Method*. Semua akan dibahas pada bab selanjutnya.

	<p>Contoh ilustrasi pada mobil. Mobil yang ada di pasaran terdiri atas berbagai tipe dan merek, tetapi semuanya memiliki <i>interface</i> kemudi yang hampir sama seperti setir, tongkat transmisi, pedal gas, dll. Jika Kita dapat mengemudikan satu jenis mobil saja dari satu merek tertentu, maka boleh dikatakan Kita dapat mengemudikan hampir semua jenis mobil yang ada karena semua mobil tersebut menggunakan <i>interface</i> yang sama. Misal, ketika Kita menekan pedal gas pada mobil A, Kita telah berhasil menggerakkan mobil A tersebut dengan percepatan yang tinggi. Sebaliknya, ketika Kita menggunakan mobil B dan sama-sama menekan pedal gas, mobil B ini bergerak agak lambat. Jadi, dapat disimpulkan dengan sama-sama Kita menekan pedal gas pada dua mobil ini akan didapatkan hasil yang berbeda pada keduanya.</p>

3.1.5 Message (Komunikasi Antar Objek)

Objek yang bertindak sendirian jarang berguna, kebanyakan objek memerlukan objek-objek lain untuk melakukan banyak hal. Oleh karena itu, objek-objek tersebut saling berkomunikasi dan berinteraksi lewat message. Ketika berkomunikasi, suatu objek mengirim pesan (dapat berupa pemanggilan *method*) untuk memberitahu agar objek lain melakukan sesuatu yang diharapkan. Seringkali pengiriman *message* juga disertai informasi untuk memperjelas apa yang dikehendaki. Informasi yang dilewatkan beserta *message* adalah parameter message.

3.2 Kelas

Kelas adalah cetak biru (rancangan) dari objek. Ini berarti Kita bisa membuat banyak objek dari satu macam kelas. Kelas mendefinisikan sebuah tipe dari objek. Di dalam kelas Kita dapat mendeklarasikan variabel dan menciptakan *Object* (instansiasi). Sebuah kelas mempunyai anggota(member) yang terdiri atas atribut dan *method*. Atribut adalah semua *field* identitas yang Kita berikan pada suatu kelas, misal kelas manusia memiliki *field* atribut berupa nama, maupun umur. *Method* dapat Kita artikan sebagai semua fungsi ataupun prosedur yang merupakan perilaku (*behaviour*) dari suatu kelas. Dikatakan fungsi bila *method* tersebut melakukan suatu proses dan mengembalikan suatu nilai (*return value*), dan dikatakan prosedur bila *method* tersebut hanya melakukan suatu proses dan tidak mengembalikan nilai (*void*).

Contoh pendeklarasian kelas:

```
class <classname>
{
    //declaration of data member
    //declaration of Methods
}
```

Kata `class` merupakan *keyword* pada Java yang digunakan untuk mendeklarasikan kelas dan `<classname>` digunakan untuk memberikan nama kelas. Sebagai contoh, dibawah ini terdapat pendeklarasian kelas Mahasiswa yang mempunyai *field* nama, nim dan *method* `getNamaMahasiswa()`.

```
class Mahasiswa
{
    String nama;           //data anggota
    int nim;               //data anggota
    void getNamaMahasiswa() {} ; //Method
}
```

Setelah mengetahui apa itu kelas , selanjutnya Kita perlu mengetahui *Object* dan *Java Modifier*.

3.3 Object

Object (objek) secara lugas dapat **diartikan sebagai insatansiasi atau hasil ciptaan dari suatu kelas** yang telah dibuat sebelumnya. Dalam pengembangan program orientasi objek lebih lanjut, sebuah objek dapat dimungkinkan terdiri atas objek-objek lain. Seperti halnya objek mobil terdiri atas mesin, ban, kerangka mobil, pintu, karoseri dan lain-lain. Atau, bisa jadi sebuah objek merupakan turunan dari objek lain sehingga mewarisi sifat-sifat induknya. Misal motor dan mobil merupakan kendaraan bermotor, sehingga motor dan mobil mempunyai sifat-sifat yang dimiliki oleh kelas kendaraan bermotor dengan spesifikasi sifat-sifat tambahan sendiri.

Contoh *Object* yang lain: Komputer, TV, mahasiswa, ponsel, lbuku, dan lainnya.

Contoh pembuatan Objek dalam java:

```
Manusia objMns1 = new Manusia(); //membuat objek manusia
```

3.4 Field

Field adalah sebuah atribut. *Field* bisa berupa sebuah variabel kelas, variabel objek, variabel *Method* objek atau parameter dari sebuah fungsi. *Field* merupakan anggota dari kelas yang digunakan untuk menyimpan data. Terdapat dua *field* dalam kelas, dibawah ini adalah contoh kelas yang terdapat 2 jenis *field* tersebut.

```
public class Circle {
    public static final double PI= 3.14159;
    public static double radiansToDegrees(double rads) {
        return rads * 180 / PI;
    }
    public double r;
    public double area() {
        return PI * r * r;
    }
    public double circumference() {
        return 2 * PI * r;
    }
}
```

1) Class Field

Field yang dikaitkan dengan kelas dimana dia didefinisikan, bukan dengan sebuah *instance* dari kelas. Di bawah ini menyatakan pendeklarasian kelas *field* pada kelas `Circle` adalah:

```
public static final double PI= 3.14159;
```

Static *modifier* diatas menyatakan bahwa *field* tersebut merupakan *class field*. *Field* ini berkaitan dengan kelas itu sendiri tidak dengan *instance* dari kelas. Berdasarkan kelas `Circle` maka jika terdapat *Method* dari kelas `Circle` maka mengakses variabel `PI` dengan `Circle.PI`

2) Instance Field

Field apapun yang dideklarasikan tanpa menggunakan *static modifier*. Contoh *instance field* yang terdapat pada kelas `Circle` adalah:

```
public double r;
```

Field ini merupakan *field* yang terkait dengan *instance* kelas bukan dengan kelas itu sendiri. Berdasarkan kelas `Circle`, setiap objek `Circle` yang dibuat memiliki salinan sendiri *field* `r`. Contohnya `r` menyatakan jari-jari lingkaran. Jadi setiap objek `Circle` dapat memiliki jari-jari sendiri dari semua objek lingkaran lainnya.

3.5 Method

Method dikenal juga sebagai suatu *function* dan *procedure*. Dalam OOP, *Method* digunakan untuk memodularisasi program melalui pemisahan tugas dalam suatu kelas. Pemanggilan *Method* menspesifikasikan nama *Method* dan menyediakan informasi (parameter) yang diperlukan untuk melaksanakan tugasnya.

Deklarasi *Method* untuk yang mengembalikan nilai (fungsi)

```
[modifier] Type-data namaMethod(parameter1,parameter2,...parameterN)
{
    Deklarasi-deklarasi dan proses ;
    return nilai-kembalian
}
```

Type-data merupakan tipe data dari nilai yang dapat dikembalikan oleh *Method* ini.

Deklarasi *Method* untuk yang tidak mengembalikan nilai (prosedur).


```
[modifier] void namaMethod(parameter1,parameter2,...parameterN)
{
    Deklarasi-deklarasi dan proses ;
}
```

Contoh implementasi *Method*:

```
public int jumlahAngka(int x, int y)
{
    int z=x+y;
    return z;
}
```

Ada dua cara melewati argumen ke *Method*, yaitu:

1) Melewatkan secara Nilai (*Pass by Value*)

Digunakan untuk argumen yang mempunyai tipe data primitif (Byte, short, int, long, float, double, char, dan boolean). Prosesnya adalah *compiler* hanya menyalin isi memori (pengalokasian suatu variabel), dan kemudian menyampaikan salinan tersebut kepada *Method*. Isi *memory* ini merupakan data “sesungguhnya” yang akan dioperasikan.

Karena hanya berupa salinan isi *memory*, maka perubahan yang terjadi pada variabel akibat proses di dalam *method* tidak akan berpengaruh pada nilai variabel asalnya.

2) Melewatkan secara Referensi (*Pass by Reference*)

Digunakan pada *array* dan objek. Prosesnya isi *memory* pada variabel *array* dan objek merupakan penunjuk ke alamat *memory* yang mengandung data sesungguhnya yang akan dioperasikan. Dengan kata lain, variabel *array* atau objek menyimpan alamat *memory* bukan isi *memory*. Akibatnya, setiap perubahan variabel di dalam *method* akan mempengaruhi nilai pada variabel asalnya.

Contoh program:

```
//file TestPass.java
class TestPass {
    int i,j;

    TestPass(int a,int b) {
        i = a;
        j = b;
    }

    //passed by value dengan parameter berupa tipe data primitif
    void calculate(int m,int n) {
        m = m*10;
        n = n/2;
    }

    //passed by reference dengan berupa tipe data class
    void calculate(TestPass e) {
        e.i = e.i*10;
        e.j = e.j/2;
    }
}
```

```
// file PassedByValue.java
class PassedByValue {
    public static void main(String[] args) {
        int x,y;
        TestPass z;
        z = new TestPass(50,100);
        x = 10;
        y = 20;
    }
}
```

```

        System.out.println("Nilai sebelum passed by value : ");
        System.out.println("x = " + x);
        System.out.println("y = " + y);

        //passed by value
        z.calculate(x,y);
        System.out.println("Nilai sesudah passed by value : ");
        System.out.println("x = " + x);
        System.out.println("y = " + y);
        System.out.println("Nilai sebelum passed by reference : ");
        System.out.println("z.i = " + z.i);
        System.out.println("z.j = " + z.j);

        //passed by reference
        z.calculate(z);

        System.out.println("Nilai sesudah passed by reference : ");
        System.out.println("z.i = " + z.i);
        System.out.println("z.j = " + z.j);
    }
}

```

Output dari program diatas

```

Nilai sebelum passed by value :
x = 10
y = 20
Nilai sesudah passed by value :
x = 10
y = 20
Nilai sebelum passed by reference :
z.i = 50
z.j = 100
Nilai sesudah passed by reference :
z.i = 500
z.j = 50

```

Keterangan:

Pada saat pemanggilan *method* `calculate()` dengan metode *pass by value*, hanya nilai dari variabel *x* dan *y* saja yang dilewatkan ke variabel *m* dan *n*, sehingga perubahan pada variabel *m* dan *n* tidak akan mengubah nilai dari variabel *x* dan *y*. Sedangkan pada saat pemanggilan *method* `calculate()` dengan metode *pass by reference* yang menerima parameter bertipe kelas `Test`. Pada waktu Kita memanggil *method* `calculate()`, nilai dari variabel *z* yang berupa referensi ke objek sesungguhnya dilewatkan ke variabel *a*, sehingga variabel *a* menunjukkan ke objek yang sama dengan yang ditunjuk oleh variabel *z* dan setiap perubahan pada objek tersebut dengan menggunakan variabel *a* akan terlihat efeknya pada variabel *z* yang terdapat pada kode yang memanggil *method* tersebut.

3.6 Keyword “this”

Dalam Java terdapat suatu besaran referensi khusus, disebut *this*, yang digunakan dalam *method* yang dirujuk untuk objek yang sedang berlaku. Nilai *this* merujuk pada objek di mana *method* yang sedang berjalan dipanggil.

Contoh implementasi program:

```

class Lagu {
    private String pencipta;
    private String judul;
    public void IsiParam(String judul,String pencipta) {
        this.judul = judul;
        this.pencipta = pencipta;
    }
}

```

```

        public void cetakKeLayar() {
            if(judul==null&& pencipta==null) return;
            System.out.println("Judul : " + judul +", pencipta : " + pencipta);
        }
    }
    class DemoLagu {
    public static void main(String[] args) {
        Lagu a = new Lagu();
        a.IsiParam("God Will Make A Way","Don Moen ");
        a.cetakKeLayar();
    }
    }
}

```

Keluaran dari program:

```
Judul : God Will Make A Way, pencipta : Don Moen
```

Keterangan :

Perhatikan pada *method* `IsiParam()` di atas. Di sana Kita mendeklarasikan nama variabel yang menjadi parameternya sama dengan nama variabel yang merupakan *property* dari kelas Lagu (judul dan pencipta). Dalam hal ini, Kita perlu menggunakan *keyword* `this` (*keyword* ini merefer ke objek itu sendiri) agar dapat mengakses *property* judul dan pencipta di dalam *method* `IsiParam()` tersebut. Apa yang terjadi jika pada *method* `IsiParam()` *keyword* `this` dihilangkan ?

Keyword `this` juga dapat digunakan untuk memanggil suatu konstruktor dari konstruktor lainnya.

Contoh Program:

```

public class Buku {
    private String pengarang;
    private String judul;
    private Buku() {
        this("Rumah Kita", "GoodBles"); // this ini digunakan untuk
                                         // memanggil konstruktor yang
                                         // menerima dua parameter
    }
    private Buku(String judul,String pengarang)
    {
        this.judul = judul;
        this.pengarang = pengarang;
    }
    private void cetakKeLayar()
    {
        System.out.println("Judul : " + judul + " Pengarang : " + pengarang);
    }
    public static void main(String[] args)
    {
        Buku a,b ;
        a = new Buku("Jurassic Park", "Michael Chricton");
        b = new Buku();
        a.cetakKeLayar();
        b.cetakKeLayar();
    }
}

```

Keluaran dari program diatas:

```
Judul : Jurassic Park Pengarang : Michael Chricton
Judul : Rumah Kita Pengarang : GoodBles
```

Pada contoh program di atas, Kita juga dapat mengambil pelajaran bahwa penggunaan *modifier private* pada *constructor* mengakibatkan *constructor* tersebut hanya dapat dikenali dalam satu kelas saja, sehingga pembuatan objek hanya dapat dilakukan dalam lokal kelas tersebut.

Catatan penting lainnya : bahwa *method* `void main(String[] args)` haruslah bersifat **public static**.



Fakultas Informatika
School of Computing
Telkom University



informatics lab

Modul 4 Encapsulation

Tujuan Praktikum

1. Memahami *Modifier*, *Constructor*, dan *Package* serta mengimplementasikannya dalam suatu program Java.

4.1 Java Modifier

Modifier memberi dampak tertentu pada kelas, *interface*, *method*, dan variabel. **Java Modifier** terbagi menjadi kelompok berikut:

- 1) **Access modifier** berlaku untuk kelas, *method* dan variabel, meliputi *modifier public*, *protected*, *private*, dan tak ada modifier.
- 2) **Final modifier** berlaku kelas, *method* dan variabel, meliputi *modifier final*.
- 3) **Static modifier** berlaku untuk variabel dan *method*, meliputi *modifier static*.
- 4) **Abstract modifier** berlaku untuk kelas dan *method*, meliputi *modifier abstract*.
- 5) **Synchronized modifier** berlaku untuk *method*, meliputi *modifier synchronized*.
- 6) **Native modifier** berlaku untuk *method*, meliputi *modifier native*.
- 7) **Storage modifier** berlaku untuk variabel, meliputi *transient* dan *volatile*.

Berikut ini *modifier-modifier* yang terdapat pada java :

Modifier	Kelas dan Interface	Method dan Variabel
(tak ada <i>modifier</i>)	Tampak di Paketnya	Diwarisi oleh subclassnya di paket yang sama dengan kelasnya. Dapat diakses oleh <i>method-method</i> di kelas-kelas yang sepaket.
Public	Tampak di manapun	Diwarisi oleh semua subclassnya. Dapat diakses dimanapun.
Protected	Tidak dapat diterapkan	Diwarisi oleh semua subclassnya. Dapat diakses oleh <i>Method-Method</i> di kelas-kelas yang sepaket.
Private	Tidak dapat diterapkan	Tidak diwarisi oleh subclassnya Tidak dapat diakses oleh kelas lain.

Permitted Modifier :

Modifier	Kelas	Interface	Method	Variabel
<i>abstract</i>	Kelas dapat berisi <i>method abstract</i> . Kelas tidak dapat diinstantiasi Tidak mempunyai <i>constructor</i>	Optional untuk diberikan di <i>interface</i> karena <i>interface</i> secara inheren adalah <i>abstract</i> .	Tidak ada badan <i>method</i> yang didefinisikan. <i>Method</i> memerlukan kelas kongkret yang merupakan subclass yang akan mengimplementasikan <i>Method abstract</i>	Tidak dapat diterapkan.
<i>final</i>	Kelas menjadi tidak dapat digunakan untuk menurunkan kelas yang baru.	Tidak dapat diterapkan.	<i>Method</i> tidak dapat ditimpa oleh <i>Method</i> di subclass-subclassnya	Berperilaku sebagai konstanta
<i>static</i>	Tidak dapat diterapkan.	Tidak dapat diterapkan.	Mendefinisikan <i>method</i> (milik) kelas. Dengan	Mendefinisikan variabel milik kelas. Tidak

Modifier	Kelas	Interface	Method	Variabel
			demikian tidak memerlukan instant <i>Object</i> untuk menjalankannya. <i>Method</i> ini tidak dapat menjalankan <i>method</i> yang bukan <i>static</i> serta tidak dapat mengacu variabel yang bukan <i>static</i> .	memerlukan instant <i>Object</i> untuk mengacunya. Variabel ini dapat digunakan bersama oleh semua <i>instant</i> objek.
<i>synchronized</i>	Tidak dapat diterapkan.	Tidak dapat diterapkan.	Eksekusi dari <i>method</i> adalah secara mutual <i>exclusive</i> diantara semua <i>thread</i> . Hanya satu <i>thread</i> pada satu saat yang dapat menjalankan <i>method</i>	Tidak dapat diterapkan pada deklarasi. Diterapkan pada instruksi untuk menjaga hanya satu <i>thread</i> yang mengacu variabel pada satu saat.
<i>native</i>	Tidak dapat diterapkan.	Tidak dapat diterapkan.	Tidak ada badan <i>method</i> yang diperlukan karena implementasi dilakukan dengan bahasan lain.	Tidak dapat diterapkan.
<i>transient</i>	Tidak dapat diterapkan.	Tidak dapat diterapkan.	Tidak dapat diterapkan.	Variabel tidak akan diserialisasi
<i>volatile</i>	Tidak dapat diterapkan.	Tidak dapat diterapkan.	Tidak dapat diterapkan.	Variabel diubah secara asinkron. Kompilator tidak pernah melakukan optimasi atasnya.

Deklarasi *modifier* dalam Java:

```
[Modifier] class/interface [nama class/interface] {} /*deklarasi modifier di class/interface*/
[Modifier][TypeData][nama Atribut]; //deklarasi modifier di atribut
[Modifier][TypeData][nama Method] (parameter_1,parameter_2,parameter_n) {}
//deklarasi modifier di Method
```

Contoh:

```
public class Manusia {} //contoh modifier di class
private int tinggi; //contoh modifier di atribut
public int getTinggi(){} //contoh modifier di Method
```

Contoh kelas yang mempunyai akses *modifier* ini ditunjukkan pada diagram kelas Manusia berikut ini.

Manusia
- nama : String - umur : int
+ setName : void + getName : String + setUmur : void + getUmur : int

Keterangan tanda:

- artinya memiliki *access modifier* **private**
- + artinya memiliki *access modifier* **public**
- # artinya memiliki *access modifier* **protected**

Contoh implementasi di dalam program:

```
public class Manusia {
    //definisi atribut
    private String nama;
    private int umur;
    //definisi Method
    public void setName(String a){
        nama=a;
    }
    public String getName(){
        return nama;
    }
    public void setUmur(int a){
        umur = a;
    }
    public int getUmur(){
        return umur;
    }
}
```

Kesepakatan umum penamaan dalam suatu Kelas:

- 1) Nama Kelas – gunakan kata benda dan huruf pertama dari tiap kata ditulis dengan huruf besar dan memiliki access modifier public : Manusia
- 2) Pada umumnya, atribut diberi access modifier private, dan method diberi access modifier public. Hal ini diterapkan untuk mendukung konsep OO yaitu enkapsulasi mengenai data hiding. Jadi, Kita tidak langsung menembak data/atribut pada kelas tersebut, tetapi Kita memberikan suatu antarmuka *method* yang akan mengakses data/atribut yang disembunyikan tersebut.
- 3) Nama atribut - gunakan kata benda, dan diawali dengan huruf kecil : nama, umur
- 4) Nama *Method* – gunakan kata kerja; kecuali huruf yang pertama, huruf awal tiap kata ditulis kapital : `getName()`, `getUmur()`
- 5) Untuk method yang akan memberikan atau mengubah nilai dari suatu atribut, nama *method* Kita tambah dengan kata kunci "set": `setUmur(int a)`, `setName(String a)`
- 6) Untuk *Method* yang akan mengambil nilai dari atribut, nama *Method* Kita tambah dengan kata kunci "get" : `getUmur()`, `getName()`
- 7) Konstanta. Semuanya ditulis dengan huruf besar; pemisah antar kata menggunakan garis bawah: `MAX_VALUE`, `DECIMAL_DIGIT_NUMBER`

4.2 Constructor

Constructor adalah tipe khusus method yang **digunakan untuk menginstansiasi atau menciptakan sebuah objek**. Nama *constructor* adalah sama dengan nama kelasnya. Selain itu, *constructor* tidak bisa mengembalikan suatu nilai (*not return value*) bahkan *void* sekalipun. *Default*-nya, bila Kita tidak membuat *constructor* secara eksplisit, maka Java akan menambahkan *constructor default* pada program yang Kita buat secara implisit. *Constructor default* ini tidak memiliki parameter masukan sama sekali. Namun, bila Kita telah mendefinisikan minimal satu buah *constructor*, maka Java tidak akan menambah *constructor default*.

Constructor juga dimanfaatkan untuk membangun suatu objek dengan langsung mengeset atribut-atribut yang disandang pada objek yang dibuat tersebut. Oleh karena itu, *constructor* jenis ini haruslah memiliki parameter masukan yang akan digunakan untuk mengeset nilai atribut.

Access modifier yang dipakai pada *constructor* selayaknya adalah *public* karena *constructor* tersebut akan diakses di luar kelasnya (walaupun Kita juga bisa memberikan *access modifier* pada *constructor* dengan *private* artinya Kita tidak bisa memanggil *constructor* tersebut di luar kelasnya). Cara memanggil *constructor* adalah dengan menambahkan *keyword new*. *Keyword new* dalam deklarasi

ini artinya Kita mengalokasikan pada memory sekian blok *memory* untuk menampung objek yang baru Kita buat.

Deklarasi *constructor*:

```
[modifier] namaclass(parameter1){
    Body constructor;
}
[modifier] namaclass(parameter1,parameter2){
    Body constructor;
}
[modifier] namaclass(parameter1,parameter2,...,parameterN){
    Body constructor;
}
```

Contoh program sebelumnya dengan penambahan *constructor* eksplisit:

```
public class Manusia {
    private String nama;
    private int umur;
    //definisi constructor
    public Manusia(){ } //constructor pertama = default tanpa parameter
    public Manusia(String a){ //constructor kedua
        nama=a;
    }
    public Manusia(String a, int b){ //constructor ketiga
        nama=a;
        umur=b;
    }
    //definisi Method
    public void setName(String a){
        nama=a;
    }
    public String getName(){
        return nama;
    }
    public void setUmur(int a){
        umur=a;
    }
    public int getUmur(){
        return umur;
    }
}

public class DemoManusia {
    public static void main(String[] args) { // program utama
        Manusia arrMns[] = new Manusia[3]; // buat array of Object
        Manusia objMns1 = new Manusia(); // constructor pertama
        objMns1.setName("Markonah");
        objMns1.setUmur(76);
        Manusia objMns2 = new Manusia("Mat Conan"); //constructor kedua
        Manusia objMns3 = new Manusia("Bajuri", 13); //constructor ketiga
        arrMns[0] = objMns1;
        arrMns[1] = objMns2;
        arrMns[2] = objMns3;

        for(int i=0; i<3; i++) {
            System.out.println("Nama : "+arrMns[i].getName());
            System.out.println("Umur : "+arrMns[i].getUmur());
            System.out.println();
        }
    }
}
```

Jika program di atas di-*compile* dan di-*running* maka *output*-nya adalah sebagai berikut.

```
Nama : Markonah
```

```
Umur : 76
Nama : Mat Conan
Umur : 0
Nama : Bajuri
Umur : 13
```

Coba analisis mengapa hasilnya seperti di atas !

Seperti *source* di atas, Kita dapat mempunyai lebih dari satu *constructor*, masing-masing harus mempunyai parameter yang berbeda sebagai penandanya. Program secara otomatis akan memilih *constructor* mana yang akan dijalankan sesuai dengan parameter masukan pada waktu pembuatan objek. Hal seperti ini disebut **Overloading** terhadap *constructor*.

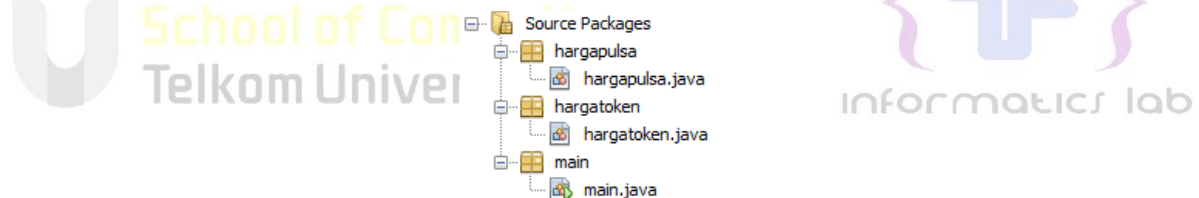
4.3 Package

Package digunakan untuk mengatur *class* yang telah dibuat dan akan bermanfaat jika *class* yang dibuat sangat banyak sehingga perlu dikelompokkan berdasarkan kategori tertentu. Selain pengelompokan *class*, penggunaan *package* dapat mengatasi konflik penamaan *method* jika suatu *method* ditulis dalam *package* yang berbeda.

Ada dua cara menggunakan suatu *package* yaitu :

- 1) *Class* yang menggunakan berada dalam direktori (*package*) yang sama dengan *Class* yang digunakan, Maka tidak diperlukan *import*.
- 2) *Class* yang menggunakan berada dalam direktori (*package*) yang berbeda dengan *Class* yang digunakan, Maka perlu dilakukan *import package*.

Dibawah ini adalah contoh penggunaan sederhana dalam *package*, terdapat 3 buah *package* dengan struktur *file* seperti berikut



Gambar 4-1 Struktur *file* program harga token dan harga pulsa.

Dengan isi tiap *class* sesuai dengan *source code* dibawah.

```
package main;
import hargatoken.hargatoken;
import hargapulsa.hargapulsa;
public class main {
    public static void main(String[] args){
        hargatoken objectToken = new hargatoken();
        objectToken.info();
        hargapulsa objectPulsa = new hargapulsa();
        objectPulsa.info();
    }
}
```

```
package hargatoken;
public class hargatoken {
    public void info(){
        System.out.println("Harga Token");
        System.out.println("Rp. 20000");
        System.out.println("Rp. 50000");
        System.out.println("Rp. 100000");
        System.out.println("Rp. 200000");
    }
}
```

```

        System.out.println("Rp. 500000");
    }
}

```

```

package hargapulsa;
public class hargapulsa {
    public void info(){
        System.out.println("Harga Pulsa");
        System.out.println("Rp. 5000");
        System.out.println("Rp. 10000");
        System.out.println("Rp. 20000");
        System.out.println("Rp. 50000");
        System.out.println("Rp. 100000");
    }
}

```

Output program:

```

Harga Token
Rp. 20000
Rp. 50000
Rp. 100000
Rp. 200000
Rp. 500000
Harga Pulsa
Rp. 5000
Rp. 10000
Rp. 20000
Rp. 50000
Rp. 100000

```

Jangan pernah
menggunakan kata
ini sebagai nama
variabel



Jangan pernah menggunakan kata-kata ini sebagai nama variabel, jika terjadi maka *compile* akan *error*

boolean	byte	char	double	float	int	long	short	public	private
protected	abstract	final	native	static	strictfp	synchronized	transient	volatile	if
else	do	while	switch	case	default	for	break	continue	assert
class	extends	implements	import	instanceof	interface	new	package	super	this
catch	finally	try	throw	throws	return	void	const	goto	enum

Modul 5 Relasi Antar Kelas

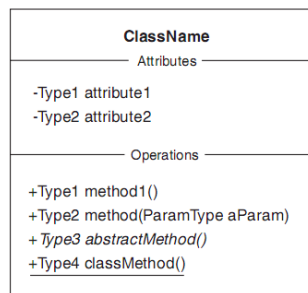
Tujuan Praktikum

2. Mengerti konsep dasar Object oriented programming
3. Memahami penggunaan kelas dalam pemrograman java
4. Memahami hubungan antar kelas baik asosiasi, agregasi, dan komposisi.

5.1 Diagram Kelas

Diagram kelas merupakan sebuah diagram yang **digunakan untuk memodelkan** kelas-kelas yang digunakan di dalam sistem beserta hubungan antar kelas dalam sistem tersebut.

Beberapa elemen penting dalam diagram kelas adalah kelas dan relasi antar kelas. Kelas digambarkan dengan simbol kotak seperti gambar di bawah ini:



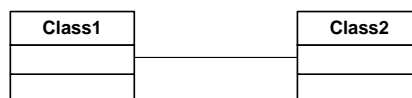
Gambar 5-1 Diagram Kelas.

Baris pertama dari simbol diagram kelas menandakan nama dari kelas yang bersangkutan. Baris di bawahnya menyatakan atribut-atribut dari kelas tersebut apa saja, dan baris setelahnya menyatakan *method-method* yang terdapat pada kelas tersebut. Adapun simbol untuk *access modifier* adalah:

Simbol	Keterangan
+	simbol <i>access modifier public</i>
-	simbol <i>access modifier private</i>
#	simbol <i>access modifier protected</i> (untuk simbol ini tidak terdapat pada gambar diatas)

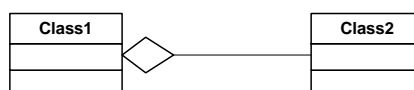
Sedangkan untuk menggambarkan **hubungan antar kelas** digunakan simbol garis antara dua kelas yang berelasi. Simbol garis tersebut antara lain:

- 1) Class1 **berasosiasi** dengan Class2 , digambarkan sebagai berikut:



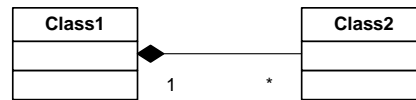
Gambar 5-2 Hubungan Asosiasi.

- 2) Class2 merupakan **elemen part-of** dari Class1 (Class1 berelasi **agregasi** dengan Class2), digambarkan sebagai berikut:



Gambar 5-3 Hubungan Agregasi.

3) Class1 dengan Class2 berelasi **komposisi**, digambarkan sebagai berikut:



Gambar 5-4 Hubungan Komposisi

4) Class1 merupakan **turunan** dari Class2, digambarkan sebagai berikut:



Gambar 5-5 Hubungan Generalisasi

5.2 Hubungan Antar Kelas

Dalam *Object Oriented Programming*, kelas-kelas yang terbentuk dapat memiliki hubungan satu dengan yang lainnya, sesuai dengan kondisi dari kelas-kelas yang bersangkutan. Ada beberapa jenis hubungan yang dapat terjadi antara kelas yang satu dengan kelas yang lainnya, antara lain:

- 1) Asosiasi
- 2) Agregasi
- 3) Komposisi
- 4) Generalisasi dan Spesialisasi (terkait dengan pewarisan)

5.2.1 Asosiasi

Asosiasi merupakan hubungan antara dua kelas terpisah yang terbentuk melalui objek pada tiap kelas. Jika dua kelas dalam suatu model perlu berkomunikasi satu sama lain, harus ada hubungan di antara kelas hal tersebut dapat diwakili oleh **asosiasi**. Contoh hubungan asosiasi ini adalah:



Gambar 5-6 Contoh hubungan asosiasi.

Pada diagram kelas di atas, terlihat hubungan antara kelas Dosen dengan kelas Mahasiswa. Method pada Kelas Mahasiswa dipanggil pada kelas Mahasiswa sehingga memiliki hubungan asosiasi.

Implementasi dari diagram kelas tersebut dalam Java adalah sebagai berikut:

File Mahasiswa.java

```
public class Mahasiswa {
    private String nim;
    private String nama;

    public void setName(String nama){
        this.nama = nama;
    }
    public void setNim(String nim){
        this.nim = nim;
    }
    public String getName(){
        return this.nama;
    }
}
```

```

    public String getNim(){
        return this.nim;
    }
}

```

File Dosen.java

```

public class Dosen {
    private String kodeDosen;
    private String namaDosen;

    //Setter
    public void setKodeDosen(String kodeDosen){
        this.kodeDosen = kodeDosen;
    }
    public void setNamaDosen(String namaDosen){
        this.namaDosen = namaDosen;
    }
    //Getter
    public String getKodeDosen(){
        return this.kodeDosen;
    }
    public String getNamaDosen(){
        return this.namaDosen;
    }

    public void giveScore(Mahasiswa s, int nilai){
        // ini asosiasi, method milik class Student dipanggil di class Dosen,
        // tp objek Student tidak menjadi atribut dr class Lecture
        s.setNilai(nilai);
    }
    public int getScore(Mahasiswa s){
        // ini asosiasi, method milik class Student dipanggil di class Lecture,
        // tp objek Student tidak menjadi atribut dr class Lecture
        return s.getNilai();
    }

    public static void main(String[] args) {
        Mahasiswa m = new Mahasiswa();

        m.setNim("130118383");
        m.setNama("Budi");

        Dosen d = new Dosen();
        d.giveScore(m, 90);

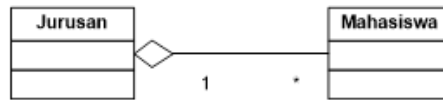
        System.out.println("Nim :"+m.getNim());
        System.out.println("Nama :"+m.getNama());
        System.out.println("Nilai :"+d.getScore(m));
    }
}

```

Pada implementasi di atas terlihat bahwa method pada kelas Mahasiswa dipanggil pada kelas Dosen pada method `getScore()` dan `giveScore()` sehingga method pada kelas Dosen menerima nilai dari objek kelas Mahasiswa.

5.2.2 Agregasi

Agregasi merupakan **hubungan** antara **dua kelas** yang **lebih ketat** dan terdapat hubungan **bagian (part)** dan **keseluruhan (whole)**. Contoh hubungan agregasi ini adalah:



Gambar 5-7 Contoh hubungan agregasi.

Pada diagram kelas di atas, terlihat hubungan antara kelas Jurusan dengan kelas Mahasiswa. Kelas mahasiswa merupakan bagian dari kelas jurusan, tetapi kelas jurusan dan kelas mahasiswa dapat diciptakan sendiri-sendiri.

Implementasi dari diagram kelas tersebut dalam Java adalah sebagai berikut:

File Mahasiswa.java

```
public class Mahasiswa {
    private String NIM, Nama;
    public Mahasiswa(String no, String nm) {
        this.NIM = no;
        this.Nama = nm;
    }

    public String GetNIM() {
        return (NIM);
    }

    public String GetNama() {
        return (Nama);
    }
}
```

File Jurusan.java

```
public class Jurusan {
    private String KodeJurusan,>NamaJurusan;
    private Mahasiswa[] Daftar = new Mahasiswa[10];
    public Jurusan(String kode, String nama) {
        this.KodeJurusan = kode;
        this>NamaJurusan = nama;
    }

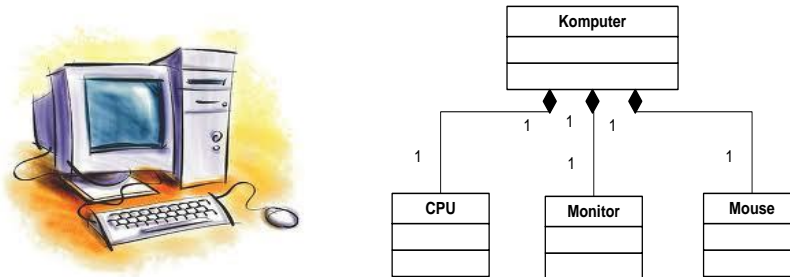
    private static int JmlMhs = 0;
    public void AddMahasiswa(Mahasiswa m) {
        this.Daftar[JmlMhs] = m;
        this.JmlMhs++;
    }

    public void DisplayMahasiswa() {
        int i;
        Sistem.out.println("Kode Jurusan : "+this.KodeJurusan);
        Sistem.out.println("Nama Jurusan : "+this>NamaJurusan);
        Sistem.out.println("Daftar Mahasiswa :");
        for (i=0;i<JmlMhs;i++)
            Sistem.out.println(Daftar[i].GetNIM()+"
"+Daftar[i].GetNama());
    }
}
```

Pada implementasi terlihat bahwa kelas jurusan memiliki atribut yang memiliki tipe kelas mahasiswa, sehingga kelas mahasiswa merupakan bagian dari kelas jurusan.

5.2.3 Komposisi

Komposisi merupakan **bentuk khusus dari agregasi** dimana terdapat hubungan antar kelas yang lebih kuat, karena yang menjadi **part (bagian)** tidak akan ada tanpa adanya kelas yang menjadi whole. Contoh hubungan komposisi adalah sebagai berikut:



Gambar 5-8 Contoh hubungan komposisi.

Dari diagram kelas di atas terlihat bahwa kelas CPU, Monitor, dan Mouse semuanya merupakan bagian dari kelas Komputer dan ketika kelas Komputer musnah maka kelas CPU, *Monitor*, dan *Mouse* akan ikut musnah. Implementasi dari diagram kelas tersebut dalam Java adalah sebagai berikut:

File CPU.java

```
public class CPU {
    private String Merk;
    private int Kecepatan;
    public CPU(String m, int k) {
        this.Merk = m;
        this.Kecepatan = k;
    }
    public void DisplaySpecCPU() {
        Sistem.out.println(this.Merk + ", " + this.Kecepatan);
    }
}
```

File Monitor.java

```
public class Monitor {
    private String Merk;
    public Monitor(String m) {
        this.Merk = m;
    }

    public void DisplaySpecMonitor() {
        Sistem.out.println(this.Merk);
    }
}
```

File Mahasiswa.java

```
//Printer.java
public class Mouse {
    private String Merk, Type;
    public Printer(String m, String t) {
        this.Merk = m;
        this.Type = t;
    }

    public void DisplaySpecMouse() {
        Sistem.out.println(this.Merk + ", " + this.Type);
    }
}
```

Pada implementasi di atas, terlihat bahwa kelas CPU, Monitor, dan Printer merupakan atribut dari kelas Komputer dan baru diciptakan pada saat instansiasi objek dari kelas Komputer.

Modul 6 Inheritance

Tujuan Praktikum

1. Mengerti dan memahami *inheritance* secara mendalam.
2. Memahami konsep *inheritance* dan *polymorfisme* serta dapat mengimplementasikan dalam suatu program.

6.1 Pendahuluan

Pada dasarnya Kita sebagai manusia sudah terbiasa untuk melihat objek yang berada di sekitar Kita tersusun secara hierarki berdasarkan kelas-nya masing-masing. Dari sini kemudian timbul suatu konsep tentang pewarisan yang merupakan suatu proses dimana suatu *class* diturunkan dari kelas lainnya sehingga ia mendapatkan ciri atau sifat dari kelas tersebut.

Dari hierarki di atas, semakin ke bawah, kelas akan semakin bersifat spesifik. Misalnya, sebuah Kelas Pria memiliki seluruh sifat yang dimiliki oleh manusia, demikian halnya juga Wanita.

Penurunan sifat tersebut dalam Bahasa Pemrograman Java disebut dengan *Inheritance* yaitu satu dalam Pilar Dasar OOP (*Object Oriented Programing*), yang dalam implementasinya merupakan sebuah hubungan “adalah bagian dari” atau “*is a relationship*” *Object* yang di-*inherit* (diturunkan).

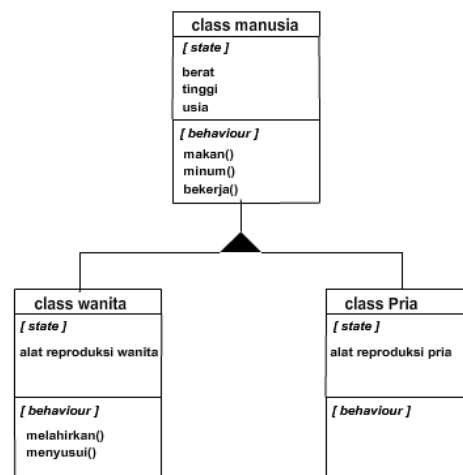
Latar belakang diperlukannya suatu *inheritance* dalam pemrograman Java adalah untuk **menghindari duplikasi *Object*** baik itu *field*, variabel maupun *method* yang sebenarnya merupakan *Object* yang bisa diturunkan dari hanya sebuah kelas. Jadi *inheritance* bukan sebuah Kelas yang diturunkan oleh sebuah *Literial*, tapi lebih menunjukkan ke hubungan antar *Object* itu sendiri.

Sedangkan suatu kemampuan dari sebuah *Object* untuk membolehkan mengambil beberapa bentuk yang berbeda agar tidak terjadi duplikasi *Object* Kita kenal sebagai *Polymorphism*.

Antara penurunan sifat (*inheritance*) maupun *Polymorphism* merupakan konsep yang memungkinkan digunakannya suatu *interface* yang sama untuk memerintah objek agar melakukan aksi atau tindakan yang mungkin secara prinsip sama namun secara proses berbeda. Dalam konsep yang lebih umum sering kali *Polymorphism* disebut dalam istilah tersebut.

6.2 Inheritance

Inheritance merupakan suatu cara untuk menurunkan suatu kelas yang lebih umum menjadi suatu kelas yang lebih spesifik. *Inheritance* adalah salah satu ciri utama suatu bahasa program yang berorientasi pada objek, dan Java pasti menggunakannya. Java mendukung *inheritance* dengan memungkinkan satu kelas untuk bersatu dengan kelas lainnya ke dalam deklarasinya. Dalam inheritance terdapat dua istilah yang sering digunakan. Kelas yang menurunkan disebut kelas dasar (*base class/super class*), sedangkan kelas yang diturunkan disebut kelas turunan (*derived class/subclass*). Karakteristik pada *superclass* akan dimiliki juga oleh *subclass*-nya.



Gambar 6-1 Ilustrasi pada kelas manusia.

Manusia adalah *superclass* dari pria dan wanita. Pria dan wanita mewarisi state dan *behaviour* kelas manusia.

Contoh pewarisan dalam program Java:

```
class A {
    int x;
    int y;
    void tampilXY() {
        System.out.println("nilai x: "+x+" nilai y: "+y);
    }
}
class B extends A {
    int z;
    void jumlahXY()
    {
        System.out.println("jumlah: "+(x+y+z));
    }
}
public class DemoInheritance {
    public static void main(String[] args) {
        A superclass=new A();
        B subclass=new B();
        System.out.println("superclass :");

        superclass.x=3;
        superclass.y=4;
        superclass.tampilXY();
        System.out.println("subclass :");

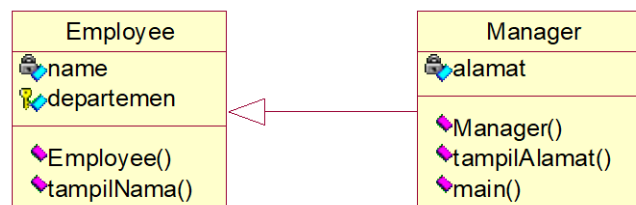
        //member superclass dapat diakses dari subclassnya
        subclass.x=1;
        subclass.y=2;
        subclass.tampilXY();

        //member tambahan hanya ada di subclass
        subclass.z=5;
        subclass.jumlahXY();
    }
}
```

Keyword Super

Keyword *super* digunakan untuk merefer *superclass* dari suatu kelas, yaitu untuk merefer *member* dari suatu *superclass*, baik atribut maupun *method*.

Contoh :



Gambar 6-2 Contoh kelas dengan *keyword super*.

Dari UML diatas dapat dilihat bahwa kelas **Employee** merupakan *superclass* dari kelas **Manager**.

```
class Employee {
    private String name;
    String departemen;
    public Employee (String s) {
        name = s;
    }
    public void tampilNama()
    {
        System.out.println("nama : "+name);
    }
}
class Manager extends Employee {
    private String alamat;

    public Manager(String nama, String s) {
        /*memanggil konstruktor employee*/
        super(nama);
        alamat = s;
    }

    public void tampilAlamat() {
        /*menginisialisasi variabel departemen yang ada pada superclass*/
        super.departemen="Personalia";

        /*memanggil Method tampilNama() yang ada pada superclass*/
        super.tampilNama();
        /*menampilkan variabel departemen yang telah diinisialisasi*/
        System.out.println("alamat : "+alamat);
        System.out.println("departemen : "+super.departemen);
    }

    public static void main(String[] args) {
        /*membuat objek*/
        Manager adi = new Manager("adi","sukabirus");
        adi.tampilAlamat();
    }
}
```

Output Program :

```
Nama : adi
Alamat : sukabirus
Departemen : Personalia
```

6.3 Overloading Method

Di dalam java Anda diperbolehkan membuat dua atau lebih *method* dengan nama yang sama dalam satu kelas, tetapi jumlah dan tipe data argumen masing – masing *method* haruslah berbeda satu dengan yang lainnya. Hal itu yang dinamakan dengan *Overloading Method*.

Contoh program:

```
class Lagu {
    String pencipta;
    String judul;

    void IsiParam(String param1) {
        judul = param1;
        pencipta = "Tidak dikenal";
    }

    void IsiParam(String param1,String param2) {
        judul = param1;
        pencipta = param2;
    }

    void CetakKeLayar() {
        System.out.println("Judul : " + judul + ", pencipta : " + pencipta);
    }
}

class DemoOver2 {
public static void main(String[] args) {
    Lagu d,e;
    d = new Lagu();
    e = new Lagu();

    d.IsiParam("Lagu 1");
    e.IsiParam("kepastian yang kutunggu","GiGi");

    d.CetakKeLayar();
    e.CetakKeLayar();
}
}
```

Pada contoh program di atas, Kita melakukan *Overloading* terhadap *method* *IsiParam()*. Di mana *IsiParam()* yang pertama menerima satu buah parameter bertipe String dan *IsiParam()* yang kedua menerima dua buah parameter bertipe String. Sehingga inilah hasil eksekusi program yang didapat:

```
Judul : Lagu 1, pencipta : Tidak dikenal
Judul : kepastian yang kutunggu: GiGi
```

6.4 Overriding Method

Di dalam java, jika dalam suatu *subclass* Anda mendefinisikan sebuah *method* yang sama dengan yang dimiliki oleh superclass, maka *method* yang Anda buat dalam *subclass* tersebut dikatakan meng-*override* superclass-nya. Sehingga jika Anda mencoba memanggil *method* tersebut dari *instance subclass* yang Anda buat, maka *method* milik *subclass*-lah yang akan dipanggil, bukan lagi *method* milik superclass-nya.

Contoh program:

```
class A {
    public void tampilkanKeLayar() {
        System.out.println("Method milik class A dipanggil...");
    }
}

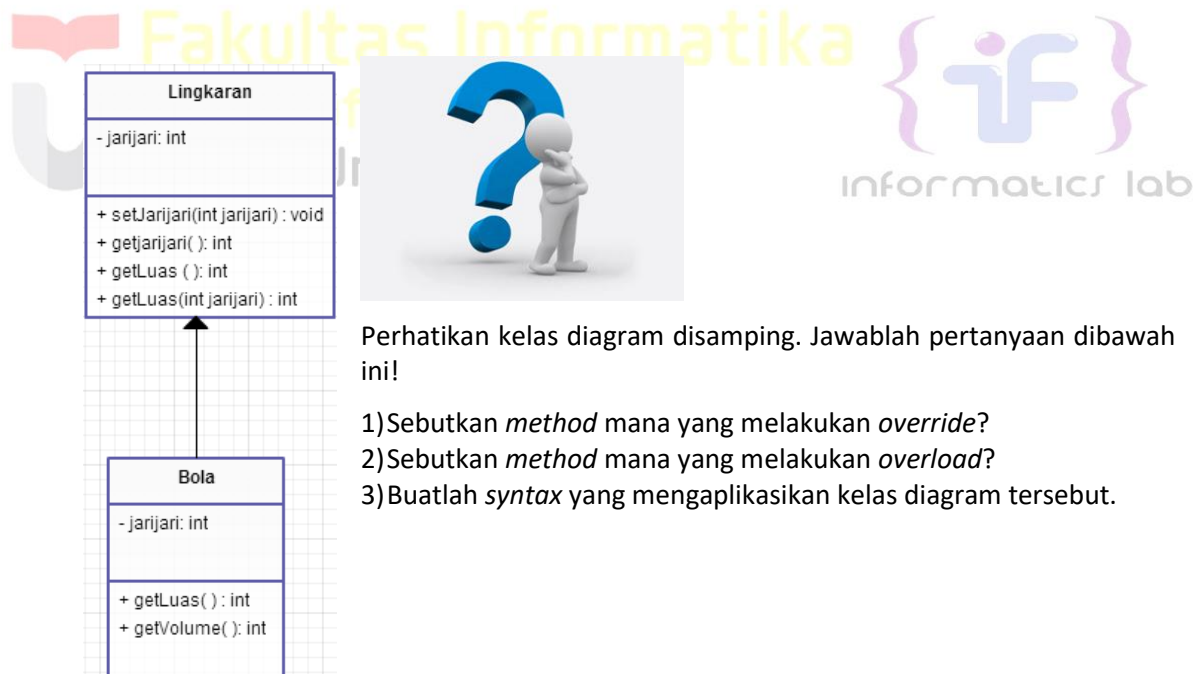
class B extends A {
    public void tampilkanKeLayar() {
        super.tampilkanKeLayar(); // Method milik superclas dipanggil
        System.out.println("Method milik class B dipanggil...");
    }
}

class DemoInheritance {
    public static void main(String[] args) {
        B subOb = new B();
        subOb.tampilkanKeLayar();
    }
}
```

Output program diatas:

```
Method milik class A dipanggil...
Method milik class B dipanggil...
```

Terlihat pada contoh di atas, **kelas B** yang merupakan turunan dari **kelas A**, meng-*override* *method* `tampilkanKeLayar()`, sehingga pada waktu Kita memanggil *method* tersebut dari variabel *instance* **kelas B** (variabel `subOb`), yang terpanggil adalah *method* yang sama yang ada pada **kelas B**.



Perhatikan kelas diagram disamping. Jawablah pertanyaan dibawah ini!

- 1)Sebutkan *method* mana yang melakukan *override*?
- 2)Sebutkan *method* mana yang melakukan *overload*?
- 3)Buatlah *syntax* yang mengaplikasikan kelas diagram tersebut.

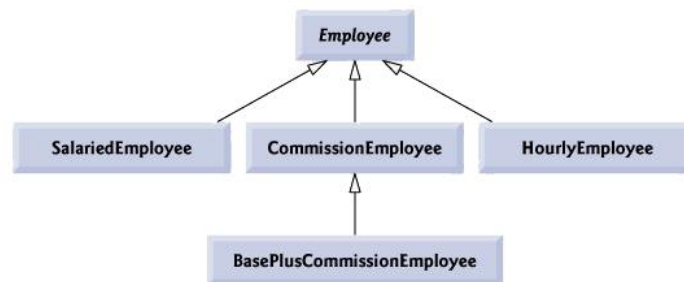
Modul 7 Abstract dan Interface

Tujuan Praktikum

1. Memahami dan mengerti konsep *interface* dan mampu mengimplementasikan dalam Java.
2. Memahami dan mengerti konsep *abstract* dan mampu mengimplementasikan dalam Java.
3. Mampu membedakan antara kelas *abstract* dan *interface*.

7.1 Kelas Abstrak

Kata kunci abstrak digunakan untuk *method* atau *class* yang **belum memiliki implementasi**. Abstrak *method* dideklarasikan pada abstrak *class*. Kelas yang dideklarasikan sebagai abstrak tidak akan bisa dibentuk *Object* dalam Java. Jadi kelas abstrak tidak akan bisa digunakan di dalam program Kita yang lain, kecuali kelas abstrak tersebut diturunkan dan turunan dari kelas abstrak tersebut yang bisa dijadikan *Object*. Sebagai ilustrasi dari kelas abstrak perhatikan gambar di bawah ini :



Gambar 7-1 Contoh kelas abstrak dalam UML.

Berdasarkan gambar diatas kelas **Employee** digunakan untuk merepresentasikan konsep umum pegawai. Kelas **Employee** diperluas ke kelas **SalariedEmployee**, **CommissionEmployee** dan **HourlyEmployee**. **BasePlusCommissionEmployee** yang memperluas **CommissionEmployee** merepresentasikan tipe akhir pegawai. Berdasarkan gambar kelas **Employee** diketik miring menandakan bahwa kelas tersebut merupakan kelas abstrak.

Berikut contoh program dari kelas abstrak *employee* dan Kelas *SalariedEmployee*.

Kelas Abstrak *Employee*

```
public abstract class Employee
{
    private String firstName;
    private String lastName;
    private String socialSecurityNumber;
    // three-argument constructor
    public Employee( String first, String last, String ssn )
    {
        firstName = first;
        lastName = last;
        socialSecurityNumber = ssn;
    } // end three-argument Employee constructor
    // set first name
    public void setFirstName( String first )
    {
        firstName = first;
    } // end Method setFirstName
    // return first name
    public String getFirstName()
    {
        return firstName;
    } // end Method getFirstName
    // set last name
    public void setLastName( String last )
    {
        lastName = last;
    } // end Method setLastName
    // return last name
    public String getLastName()
    {
        return lastName;
    } // end Method getLastName
    // set social security number
    public void setSocialSecurityNumber( String ssn )
    {
        socialSecurityNumber = ssn; // should validate
    } // end Method setSocialSecurityNumber

    // return social security number
    public String getSocialSecurityNumber()
    {
        return socialSecurityNumber;
    } // end Method getSocialSecurityNumber

    // return String representation of Employee Object
    public String toString()
    {
        return String.format( "%s %s\nsocial security number: %s",
                               getFirstName(), getLastName(), getSocialSecurityNumber() );
    } // end Method toString

    // abstract Method overridden by subclasses
    public abstract double earnings(); // no implementation here
} // end abstract class Employee
```

Kelas *SalariedEmployee* yang merupakan turunan dari Kelas Abstrak.

```
public class SalariedEmployee extends Employee {

    private double weeklySalary;

    // four-argument constructor
    public SalariedEmployee( String first, String last, String ssn,
        double salary )
    {
        super( first, last, ssn ); // pass to Employee constructor
        setWeeklySalary( salary ); // validate and store salary
    } // end four-argument SalariedEmployee constructor

    // set salary
    public void setWeeklySalary( double salary )
    {
        weeklySalary = salary < 0.0 ? 0.0 : salary;
    } // end Method setWeeklySalary

    // return salary
    public double getWeeklySalary()
    {
        return weeklySalary;
    } // end Method getWeeklySalary

    // calculate earnings; override abstract Method earnings in Employee
    public double earnings()
    {
        return getWeeklySalary();
    } // end Method earnings

    // return String representation of SalariedEmployee Object
    public String toString()
    {
        return String.format( "salaried employee: %s\n%s: $%,.2f",
            super.toString(), "weekly salary", getWeeklySalary() );
    } // end Method toString
} // end class SalariedEmployee
```

7.2 Interface

Interface serupa dengan kelas. Kelas mendefinisikan kemampuan sebuah *Object*, sementara *interface* mendefinisikan method atau konstanta yang akan diimplementasikan pada *Object* yang lain. *Interface* membantu mendefinisikan sifat *Object* dengan mendeklarasikan seperangkat karakteristik *Object* tersebut. Sebagai contoh kasus, radio, TV, dan *speaker* komputer memiliki pengontrol volume. Untuk alasan ini, mungkin Kita menginginkan *device-device* ini mengimplementasikan *interface* yang bernama `VolumeControl()`.

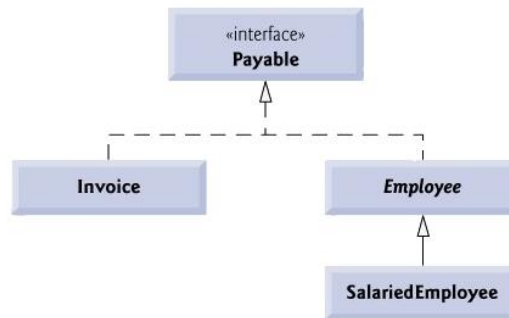
Interface pada Java 8 memiliki beberapa batasan:

- 1) Semua atribut adalah *public*, *static* dan *final* (semua atribut bertindak sebagai konstanta)
- 2) Hanya memiliki *default method*, dan *static method*
- 3) Tidak boleh ada deklarasi konstruktor.

Berbeda dengan *class* biasa, *interface* mengizinkan **multiple inheritance**, yaitu sebuah kelas dapat mempunyai dua *interface* induk sekaligus. Hal tersebut tidak dapat dilakukan pada sebuah kelas, dimana sebuah kelas hanya boleh mempunyai satu kelas induk.

Syntax yang diperlukan hampir sama dengan cara **membuat kelas**. Perbedaannya hanya *method* pada **interface tidak memiliki isi/ variabel**. Sebuah contoh dengan tiga item: *interface*, kelas yang

mengimplementasikan *interface*, dan kelas yang menggunakan kelas yang disalurkan. Sebagai ilustrasi perhatikan gambar UML berikut:



Gambar 7-2 Contoh kelas *interface* dalam UML.

Berdasarkan gambar diatas hierarki dimulai dengan *interface Payable*. Relasi UML yang terlihat antara kelas dan interface dinamakan **realization**. Sebuah kelas dikatakan merealisasikan atau mengimplementasikan method pada sebuah **interface**.

Berikut contoh program dari UML diatas (Hanya *interface Payable* dan Kelas Invoice) .

Interface Payable

```

public interface Payable
{
    double getPaymentAmount(); // calculate payment; no implementation
} // end interface Payable
    
```

Kelas Invoice yang mengimplementasikan Payable

```

public class Invoice implements Payable
{
    private String partNumber;
    private String partDescription;
    private int quantity;
    private double pricePerItem;

    // four-argument constructor
    public Invoice( String part, String description, int count,
        double price )
    {
        partNumber = part;
        partDescription = description;
        setQuantity( count ); // validate and store quantity
        setPricePerItem( price ); // validate and store price per item
    } // end four-argument Invoice constructor

    // set part number
    public void setPartNumber( String part )
    {
        partNumber = part;
    } // end Method setPartNumber

    // get part number
    public String getPartNumber()
    {
        return partNumber;
    } // end Method getPartNumber

    // set description
    public void setPartDescription( String description )
    {
        partDescription = description;
    } // end Method setPartDescription
}
    
```

```

// get description
public String getPartDescription()
{
    return partDescription;
} // end Method getPartDescription

// set quantity
public void setQuantity( int count )
{
    quantity = ( count < 0 ) ? 0 : count; // quantity cannot be negative
} // end Method setQuantity

// get quantity
public int getQuantity()
{
    return quantity;
} // end Method getQuantity

// set price per item
public void setPricePerItem( double price )
{
    pricePerItem = ( price < 0.0 ) ? 0.0 : price; // validate price
} // end Method setPricePerItem

// get price per item
public double getPricePerItem()
{
    return pricePerItem;
} // end Method getPricePerItem

// return String representation of Invoice Object
public String toString()
{
    return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
        "invoice", "part number", getPartNumber(), getPartDescription(),
        "quantity", getQuantity(), "price per item", getPricePerItem() );
} // end Method toString

// Method required to carry out contract with interface Payable
public double getPaymentAmount()
{
    return getQuantity() * getPricePerItem(); // calculate total cost
} // end Method getPaymentAmount
} // end class Invoice

```

```

public interface Product {
    static final String MAKER = "My Corp";
    static final String PHONE = "555-123-4567";
    public int getPrice(int id);
}

public class Shoe implements Product {
    public int getPrice(int id) {
        if (id == 1)
            return(5);
        else
            return(10);
    }

    public String getMaker() {
        return(MAKER);
    }
}

```

```

public class Store {
    static Shoe hightop;
    public static void init() {
        hightop = new Shoe();
    }
}

public static void main(String argv[]) {
    init();
    getInfo(hightop);
    orderInfo(hightop);

    public static void getInfo(Shoe item) {
        System.out.println("This Product is made by " + item.MAKER);
        System.out.println("It costs $" + item.getPrice(1) + '\n');
    }

    public static void orderInfo(Product item) {
        System.out.println("To order from "+item.MAKER+" call "+item.PHONE+".");
        System.out.println("Each item costs $" + item.getPrice(1));
    }
}

```

Interface sebelum Java 8 hanya dapat memiliki *method* abstrak. Semua *method* di *Interface* bersifat publik dan abstrak secara *default*. Namun pada Java 8 memungkinkan *interface* untuk memiliki *method default* dan *static*. Hal tersebut dikarenakan *default method* pada *interface* dapat memungkinkan pengembang menambahkan *method* baru ke *interface* tanpa mempengaruhi kelas yang mengimplementasikan *interface*.

Default method pada *Interface* :

```

interface MyInterface{
    default void newMethod(){
        System.out.println("Newly added default method");
    }
    void existingMethod(String str);
}

public class Example implements MyInterface{
    // implementasi abstract method
    public void existingMethod(String str){
        System.out.println("String is: "+str);
    }
    public static void main(String[] args) {
        Example obj = new Example();
        obj.newMethod();
        obj.existingMethod("Java 8 is easy to learn");
    }
}

```

Method **newMethod** pada *interface* **MyInterface** yang berarti kita tidak perlu mengimplementasikan *method* tersebut di kelas implementasi **Example**. Dengan menggunakan metode ini kita dapat menambahkan *method default* ke dalam *interface* tanpa perlu khawatir dengan kelas yang mengimplementasikan *interface* tersebut.

Static method pada *Interface* :

```
interface MyInterface{
    default void newMethod(){
        System.out.println("Newly added default method");
    }
    static void anotherNewMethod(){
        System.out.println("Newly added static method");
    }
    void existingMethod(String str);
}
public class Example implements MyInterface{
    // implementing abstract method
    public void existingMethod(String str){
        System.out.println("String is: "+str);
    }
    public static void main(String[] args) {
        Example obj = new Example();

        obj.newMethod();
        MyInterface.anotherNewMethod();
        obj.existingMethod("Java 8 is easy to learn");
    }
}
```

Static method pada *interface* hampir sama dengan *default method* sehingga kita tidak perlu mengimplementasikannya di kelas **Example**. Kita dapat dengan aman menambahkannya ke *interface* yang ada tanpa mengubah kode di kelas **Example**. Karena *static method* tidak dapat menyimpannya di kelas **Example**.

Perbedaan **abstract class** dan **interface** antara lain:

- 1) Semua *interface* *Method* tidak memiliki *body* sedangkan beberapa *abstract class* dapat memiliki *Method* dengan implementasi.
- 2) Sebuah *interface* hanya dapat mendefinisikan *constant* sedangkan sebuah *abstract class* tampak seperti kelas biasa yang dapat mendeklarasikan variabel.
- 3) *Interface* tidak memiliki hubungan *inheritance* secara langsung dengan sebuah *kelas* tertentu, sedangkan *abstract class* bisa jadi hasil turunan dari *abstract class* induknya.
- 4) *Interface* memungkinkan terjadinya pewarisan jamak (*multiple inheritance*) sedangkan *abstract class* tidak.

Modul 8 Polymorphism

Tujuan Praktikum

1. Memahami dan mengerti konsep *polymorphism* dan mampu mengimplementasikan dalam Java.

8.1 Pendahuluan

Polymorphism berasal dari bahasa Yunani yang berarti banyak bentuk. Dalam PBO, konsep ini memungkinkan digunakannya suatu *interface* yang sama untuk memerintah objek agar melakukan aksi atau tindakan yang mungkin secara prinsip sama namun secara proses berbeda.

Dalam Konsep yang lebih umum sering kali *Polymorphism* disebut dalam istilah satu *interface* banyak aksi. Contoh yang konkrit dalam dunia nyata yaitu mobil.

Mobil yang ada dipasaran terdiri atas berbagai tipe dan berbagai merk, namun semuanya memiliki *interface* kemudi yang sama, seperti: stir, tongkat transmisi, pedal gas dan rem. Jika seseorang dapat mengemudikan satu jenis mobil saja dari satu merk tertentu, maka orang itu akan dapat mengemudikan hampir semua jenis mobil yang ada, karena semua mobil tersebut menggunakan *interface* yang sama.

Interface yang sama tidak berarti cara kerjanya juga sama. Misal pedal gas, jika ditekan maka kecepatan mobil akan meningkat, tapi proses peningkatan kecepatan ini berbeda-beda untuk setiap jenis mobil. Dengan OOP maka dalam melakukan pemecahan suatu masalah Kita tidak melihat bagaimana cara menyelesaikan suatu masalah tersebut (terstruktur) tetapi objek-objek apa yang dapat melakukan pemecahan masalah tersebut.

Sebagai contoh anggap Kita memiliki departemen yang memiliki manager, sekretaris, petugas administrasi data dan lainnya. Misal manager ingin memperoleh data dari bag administrasi maka manager tersebut dapat menyuruh petugas bag administrasi untuk mengambilnya. Pada kasus tersebut, manager tidak harus mengetahui bagaimana cara mengambil data tetapi manager bisa mendapatkan data tersebut melalui objek petugas administrasi.

Jadi untuk menyelesaikan suatu masalah dengan kolaborasi antar objek-objek yang ada karena setiap objek memiliki deskripsi tugasnya sendiri.

8.2 Referensi Variabel Casting

Terdapat 2 tipe variabel reference yaitu **Downcasting** dan **Upcasting**.

Downcasting: apabila terdapat variabel *reference* yang mengacu pada sub tipe objek, maka dapat dimasukkan ke dalam sub tipe variabel *reference*. Dengan kata lain **downcasting** hanya dapat dilakukan bila antara type dari *Object* dan type dari variabel *reference* masih dalam sebuah inheritance. Bila ada *Polymorphisme* sebagai berikut:

```
A ref = new B();  
// Dimana B adalah subclass dari A, maka dari variabel ref Kita hanya akan dapat  
meng-invoke (menjalankan) Method-Method yang dideklarasikan di kelas A. Bila Kita  
ingin meng-invoke Method yang hanya terdapat di kelas B, maka Kita tidak dapat  
secara langsung memanggil Method yang hanya ada di B seperti sebagai berikut:  
ref.MethodDiB(); //COMPILE TIME ERROR !!!!
```

Potongan program di atas akan menghasilkan *compile time error* sbb:

```
Cannot find symbol
```

Agar dapat **meng-*invoke* method** yang ada di **kelas B**, Kita dapat menggunakan **casting** seperti sebagai berikut:

```
//cara 1
B b = (B) ref;    //CASTING !!!!
b.MethodDiB();
//Atau :
//cara 2
((B)ref).MethodDiB(); //CASTING !!!!
```

Contoh :

```
class A {
    void MethodDiA() {
        System.out.println("A.MethodDiA()");
    }
}
class B extends A {
    void MethodDiB() {
        System.out.println("B.MethodDiB()");
    }
}
class Polymorphisme01 {
    public static void main(String[] args) {
        A ref = new B();
        //cara 1
        B b = (B) ref;
        b.MethodDiB();
        //cara 2
        ((B)ref).MethodDiB();
    }
}
```

Program di atas akan menghasilkan :

```
B.MethodDiB()
B.MethodDiB()
```

Downcasting hanya dapat dilakukan bila **antara type** dari *Object* dan *type* dari variabel *reference* masih dalam sebuah **inheritance**. Bila tidak, maka akan terjadi *compile time error (inconvertible types)*. Contoh:

```
class A {}
class B extends A {}
class C {}

class Test {
    void lAkukanSesuatu() {
        A ab = new B();
        B b = (B) ab;    //Downcast berhasil
        C c = (C) ab;    //Downcast GAGAL
    }
}
```

Compile time error yang dihasilkan adalah :

inconvertible types

Ada kemungkinan *downcast* tidak *error* saat *compile time*, akan tetapi melemparkan **ClassCastException**. Hal ini terjadi karena *type* dari *Object* yang di-*assign* ke suatu variabel *reference* adalah *superclass* dari *type* dari variabel *references* tersebut.

Contoh:

```
class A {}
class B extends A {}

class Test {
    public static void main(String[] args) {
        Test t = new Test();
        t.lakukanSesuatu();
    }

    void lakukanSesuatu() {
        A a = new A();
        /*Statement di bawah dapat compile, akan tetapi
        *saat dijalankan akan melemparkan exception
        *ClassCastException !!!!
        */
        B b = (B) a;
    }
}

/*
Saat compile berhasil !!! (dengan javac)
Saat dijalankan (dengan java) akan menghasilkan :
java.lang.NoClassDefFoundError: inheritance/VarRefCast03
Exception in thread "main"
Java Result: 1
*/
```

Kesimpulan untuk *downcast* adalah :

- 1) *Downcast* pasti eksplisit.
- 2) *Downcast* akan berhasil saat *compile time* bila antara *Object* dengan variabel *reference* masih dalam 1 buah jalur *inheritance*.
- 3) *Downcast* akan gagal saat *runtime* (melemparkan *exception*) bila ternyata *type* dari *Object* yang di-assign ke variabel *reference* ternyata adalah *superclass* dari variabel *reference* itu.

Upcasting: mengisikan variabel *reference* ke variabel *reference* lainnya dengan tipe superclassnya.

Upcasting bisa dilakukan dengan dua cara :

- 1) **Implisit upcasting** (otomatis)
- 2) **Eksplisit upcasting**

Contoh :

```
class A {}
class B extends A {}
class VarRefCast01 {
    public static void main(String[] args) {
        B b = new B();
        //implisit upcasting
        //dari variabel reference ke variabel reference lainnya
        A ab1 = b;
        //implisit upcasting
        //dari suatu Object ke variabel reference
        A ab2 = new B();
        //eksplisit upcasting
        //dari variabel reference ke variabel reference lainnya
        A ab3 = (A) b;
        //eksplisit upcasting
        //dari Object ke variabel reference
        A ab4 = (A) new B();
    }
}
```

Modul 9 Inner Class, Collection, dan Generics

Tujuan Praktikum

1. Menjelaskan pengertian *inner class* dan menjelaskan penggunaannya.
2. Menjelaskan pengertian *static variable* dan *static method* dan menjelaskan penggunaannya.
3. Menjelaskan pengertian collection dan menjelaskan jenis-jenis collection serta perbedaan penggunaannya.
4. Menjelaskan pengertian generics dan menjelaskan penggunaan generics.

9.1 Inner Class

Inner class adalah kelas yang didefinisikan di dalam sebuah kelas lainnya. Dengan didefinisikan di dalam sebuah kelas, *inner class* memiliki akses khusus ke kelas yang melingkupinya. *Inner class* bahkan dapat mengakses atribut/*method* dari kelas yang melingkupinya, walaupun bersifat *private*. Contoh penggunaan *inner class* sebagai berikut:

```
public class MyOuter{
    private int x = 0;
    private MyInner inner;
    public void outerMethod(){
        inner = new MyInner();
        inner.innerMethod();
    }
    public int getX(){
        return x;
    }
    public class MyInner{
        public void innerMethod(){
            x = 10;
        }
    }
}

public class Driver{
    public static void main(String args[]){
        MyOuter outer = new MyOuter();
        outer.outerMethod();
        System.out.println(outer.getX());
    }
}
```

Objek dari *inner class* tidak bisa berdiri sendiri. Untuk membentuk objek dari *inner class*, dibutuhkan objek dari *outer class* terlebih dahulu.

Terdapat tiga cara yang dapat digunakan untuk mendefinisikan *inner class*.

- 1) Definisi *inner class* di dalam *body class*

```
// file: OuterDemo.java
public class OuterDemo {
    private int num;
    class InnerDemo {
        public void print() {
            System.out.println("This is an inner class");
        }
        public int getNum() {
            return num;
        }
    }
}
```



```
// file: MyClass1.java
public class MyClass1 {
    public static void main(String[] args) {
        OuterDemo.InnerDemo inner = new OuterDemo().new InnerDemo();
        inner.print();
        System.out.println("Num = "+inner.getNum());
    }
}
```

Cara pertama adalah dengan mendefinisikan *inner class* di dalam *body class*. Pemanggilan objek dari *inner class* harus dilakukan dari objek kelas yang melingkupinya. Seperti pada contoh di atas, untuk memanggil objek dari kelas **InnerDemo** harus dilakukan pemanggilan objek **OuterDemo** terlebih dahulu.

2) Definisi *inner class* di dalam *body method*

```
// file: OuterClass.java
public class OuterClass {
    void myMethod() {
        int num = 23;
        class MethodInnerDemo {
            public void print() {
                System.out.println("The number = "+num);
            }
        }
        MethodInnerDemo mid = new MethodInnerDemo();
        mid.print();
    }
    public static void main(String[] args) {
        OuterClass oc = new OuterClass();
        oc.myMethod();
    }
}
```

Cara kedua adalah dengan mendefinisikan *inner class* di dalam *body method*. Hal ini sering disebut juga dengan *local class*. Karena kelas **MethodInnerDemo** didefinisikan di dalam *Method myMethod()*, maka kelas tersebut hanya dikenali di dalam *method myMethod()* dan tidak dapat didefinisikan di luar *method myMethod()*.

3) Definisi *inner class* menggunakan *anonymous class*

Cara ketiga mendefinisikan *inner class* adalah menggunakan **anonymous class**. **Anonymous class** adalah kelas yang tidak didefinisikan namanya. Contoh penggunaannya sebagai berikut.

```
// file: OuterClass2.java
abstract class AnonymousInner {
    public abstract void myMethod();
}

public class OuterClass2 {
    public static void main(String[] args) {
        AnonymousInner inner = new AnonymousInner {
            @Override
            public void myMethod() {
                System.out.println("Anonymous class sample");
            }
        }
        inner.myMethod();
    }
}
```

Perhatikan kode di atas. Terlihat bahwa objek *inner* yang dipanggil di dalam *method main* adalah objek yang berasal dari penurunan kelas abstrak **AnonymousInner**. Namun kelas

yang menurunkannya tidak diberikan nama, karena pendefinisannya langsung menggunakan *constructor AnonymousInner* dengan meng-*override method* abstrak.

Cara lain untuk mendefinisikan **anonymous class** adalah sebagai berikut.

```
// file: MyClass2.java
interface Message {
    String greet();
}

public class MyClass2 {
    public void displayMessage(Message m) {
        System.out.println(m.greet()+" , how do you do?");
    }
    public static void main(String[] args) {
        MyClass2 obj = new MyClass2();
        obj.displayMessage(new Message(){
            @Override
            public String greet() {
                return "Hello";
            }
        });
    }
}
```

Pada contoh di atas, pendefinisian *anonymous class* dilakukan pada saat pemanggilan *method displayMessage()* oleh objek obj. pendefinisian dilakukan di dalam parameter *method displayMessage()*.

9.2 Static

9.2.1 Static Variabel

Static variabel yaitu pendefinisian variabel milik kelas yang tidak memerlukan *instant object* untuk mengacunya. Variabel ini dapat digunakan bersama oleh semua *instant object*. Static variabel juga biasa disebut variabel *class*, serta *local* variabel tidak dapat didefinisikan sebagai static.

Contoh penggunaan static variabel sebagai berikut:

```
public class Share{
    private int privateInt;
    private static int staticInt;
    public Share(int pr, int si){
        privateInt = pr;
        staticInt = si;
    }
    public String toString(){
        return privateInt +""+ staticInt;
    }
}

public class Driver{
    public static void main(String args[]){
        Share s1 = new Share(4,4);
        System.out.println(s1.toString());
        Share s2 = new Share(8,2);
        System.out.println(s1.toString());
        System.out.println(s2.toString());
        Share s3 = new Share(6,22);
        System.out.println(s1.toString());
        System.out.println(s2.toString());
        System.out.println(s3.toString());
    }
}
```

Output :

```
> 4 4
> 4 2
> 8 2

> 4 22
> 8 22
> 6 22
```

9.2.2 Static Method

Static *method* merupakan pendefinisian *method* (milik) kelas, dengan demikian tidak memerlukan *instant object* untuk menjalankannya. *Method* ini tidak dapat menjalankan *method* yang bukan static serta tidak dapat mengacu variabel yang bukan static.

Contoh penggunaan static *method* adalah sebagai berikut :

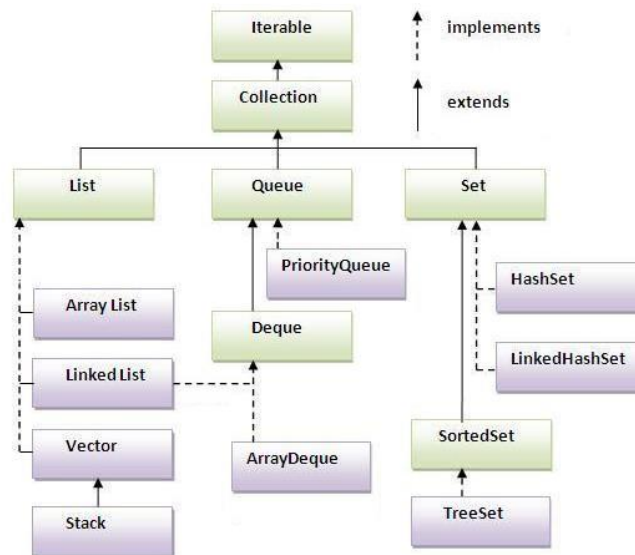
```
public class CounterMachine{
    public static void count(){
        counter++;
    }
}

public class Driver{
    public static void main(String args[]){
        for(int i = 0; i < 10; i++){
            if( i % 2 == 0){
                CounterMachine.count();
            }
        }
        System.out.println(CounterMachine.counter);
    }
}
```

9.3 Collection

Collection adalah objek khusus yang digunakan untuk menampung banyak objek yang bertipe apapun. *Collection* dapat menampung banyak objek bertipe *Object*. Seperti diketahui, semua kelas di Java merupakan turunan dari kelas *Object*, maka dengan prinsip polimorfisme, *collection* dapat menampung objek bertipe apapun.

Collection didefinisikan sebagai *interface*. Kita dapat menggunakan kelas turunan dari *Collection*. Contoh hierarki yang ada pada *Collection* dapat dilihat pada Gambar 9-1.



Gambar 9-1 Hierarki dari *Collection*.

Dengan menggunakan *generics*, Kita dapat membatasi tipe yang dapat ditampung oleh *Collection*. Contohnya sebagai berikut.

```
Collection<String> str = new HashSet<String>();
List<Integer> arr_i = new List ();
ArrayList<Employee> emp = new ArrayList ();
```

Beberapa *method* yang ada pada *Collection* antara lain:

Table 9-1 Beberapa *Method Collection*.

Nama Method	Deskripsi
<code>boolean add(Object item)</code>	Menambahkan item ke dalam <i>collection</i>
<code>boolean remove(Object item)</code>	Menghapus item dari dalam <i>collection</i>
<code>boolean addAll(Collection c)</code>	Menambahkan semua item dari <i>c</i> ke dalam <i>collection</i>
<code>boolean removeAll(Collection c)</code>	Menghapus semua item yang ada pada <i>c</i> dari dalam <i>collection</i>
<code>boolean retainAll(Collection c)</code>	Menghapus semua item yang tidak ada pada <i>c</i> dari dalam <i>collection</i>
<code>boolean contains(Object item)</code>	Mengecek apakah item ada di dalam <i>collection</i>

Terdapat beberapa cara yang dapat digunakan untuk mengakses semua elemen dari *Collection*.

```
import java.util.*;

public class JavaApplication1 {
    public static void main(String[] args) {
        List list = new ArrayList();
        list.add(200);
        list.add("Hello");
        list.add(235);
        list.add("Donny");

        // normal loop
        System.out.println("Normal loop");
        for (int i = 0; i < list.size(); i++) {
            Object o = list.get(i);
            System.out.println(o);
        }

        // loop using for-element
        System.out.println("for-element loop");
    }
}
```

```

for (Object o : list) {
    System.out.println(o);
}

// Loop using iterator
System.out.println("loop using iterator");
Iterator itr = list.iterator();
while (itr.hasNext()) {
    Object o = itr.next();
    System.out.println(o);
}

// loop using lambda Expression
System.out.println("loop using lambda expression");
list.forEach(o -> System.out.println(o));

// loop using reference
System.out.println("loop using reference");
list.forEach(System.out::println);
}
}

```

9.3.1 Melakukan Sorting pada Collection

Untuk melakukan *sorting* pada *Collection*, Kita dapat menggunakan dua acara:

- 1) Menggunakan *interface Comparable*

Untuk melakukan *sorting*, dapat menggunakan *method* `Collections.sort(list)` atau `list.sort()`. Jika *list* mengandung elemen bertipe *String*, maka *list* akan diurutkan berdasarkan urutan abjad. Jika *list* mengandung elemen bertipe *Date*, maka *list* akan diurutkan berdasarkan urutan tanggal. Untuk membangun pengurutan sendiri pada *list* bertipe kelas tertentu, maka kelas tersebut harus mengimplementasikan *interface Comparable* dan meng-*override* *method* `compareTo(Object obj)`. *Method* `compareTo(Object obj)` akan mengembalikan:

- a. Nilai < 0, jika objek yang memanggil diurutkan sebelum objek obj
- b. Nilai = 0, jika objek yang memanggil sama dengan objek obj
- c. Nilai > 0, jika objek yang memanggil diurutkan setelah objek obj

Jika parameter pada `compareTo()` adalah *String* atau *Date*, maka pada `compareTo()` panggil:

```
return this.getString().compareTo(obj.getString())
```

Jika parameter pada `compareTo()` adalah angka, maka pada `compareTo()` melakukan pengurangan, seperti:

```
return this.getNumber() - obj.getNumber()
```

Contoh penggunaan *interface Comparable* sebagai berikut.

```

import java.util.*;
public class Employee implements Comparable<Employee> {
    private String name;
    private int salary;
    public Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }
    public String getName() {
        return name;
    }
}

```

```

    public int getSalary() {
        return salary;
    }
    @Override
    public String toString() {
        return "name=" + name + ", salary=" + salary;
    }
    @Override
    public int compareTo(Employee e) {
        return name.compareTo(e.name);
    }
    public static void main(String[] args) {
        List<Employee> listEmp = new ArrayList();
        listEmp.add(new Employee("bobby", 3000));
        listEmp.add(new Employee("erick", 1600));
        listEmp.add(new Employee("rey", 2500));
        listEmp.add(new Employee("anna", 3500));
        Collections.sort(listEmp);
        listEmp.forEach(System.out::println);
    }
}

```

Output yang muncul sebagai berikut:

```

name=anna, salary=3500.0
name=bobby, salary=3000.0
name=erick, salary=1600.0
name=rey, salary=2500.0

```

2) Menggunakan *interface* Comparator<T>

Interface comparable memungkinkan untuk melakukan *sorting* pada satu *property* saja. Untuk melakukan *sorting* pada *multiple property*, gunakan *interface* Comparator. Caranya adalah dengan membuat kelas yang mengimplementasikan *interface* Comparator<T> dan meng-*override* Method *compare*(T obj1, T obj2). Contohnya Kita dapat membangun kelas *comparator<>* untuk kelas *Employee()* sebagai berikut:

```

import java.util.*;
public class SalaryComparator implements Comparator<Employee> {
    @Override
    public int compare(Employee e1, Employee e2) {
        return e1.getSalary() - e2.getSalary();
    }
}

```

Penggunaan kelas Comparator pada *method* main sebagai berikut.

```

public static void main(String[] args) {
    List<Employee> listEmp = new ArrayList();
    listEmp.add(new Employee("bobby", 3000));
    listEmp.add(new Employee("erick", 1600));
    listEmp.add(new Employee("rey", 2500));
    listEmp.add(new Employee("anna", 3500));
    Collections.sort(listEmp);
    System.out.println("Sorted by name");
    listEmp.forEach(System.out::println);
    Collections.sort(listEmp, new SalaryComparator());
    System.out.println("Sorted by salary");
    listEmp.forEach(System.out::println);
}

```

Output yang dihasilkan sebagai berikut.

```

Sorted by name
name=anna, salary=3500
name=bobby, salary=3000
name=erick, salary=1600

```

```
name=rey, salary=2500
Sorted by salary
name=erick, salary=1600
name=rey, salary=2500
name=bobby, salary=3000
name=anna, salary=3500
```

9.3.2 Melakukan Filtering pada Collection

Untuk melakukan *filtering* pada *Collection*, Kita dapat menggunakan dua cara. Misalkan Kita ingin melakukan *filtering* pada *list Employee* berdasarkan kriteria tertentu. Misalkan Kita ingin mencari daftar *Employee* yang memiliki gaji minimal 3000. Maka, Kita dapat menggunakan dua cara.

- 1) Menggunakan cara konvensional seperti berikut.

```
System.out.println("Employee with salary minimal 3000");
for(Employee e: listEmp) {
    if(e.getSalary() >= 3000) {
        System.out.println(e);
    }
}
```

- 2) Menggunakan ekspresi lambda seperti berikut.

```
System.out.println("Employee with salary minimal 3000");
List<Employee> temp = listEmp.Stream().
    filter(e -> e.getSalary() >= 3000).collect(Collectors.toList());
temp.forEach(System.out::println);
```

Atau Kita ingin mencari **Employee** dengan nama tertentu. Maka Kita dapat menggunakan dua cara:

- 1) Menggunakan cara konvensional seperti berikut.

```
System.out.println("Employee erick");
for(Employee e: listEmp) {
    if(e.getName().equals("erick")) {
        System.out.println(e);
    }
}
```

- 2) Menggunakan ekspresi lambda seperti berikut.

```
System.out.println("Employee erick");
System.out.println(listEmp.Stream().
    filter(e -> e.getName().equals("erick")).
    findFirst().orElse(null));
```

9.4 Generics

Generic adalah sebuah cara untuk memberikan batasan tipe kepada sebuah kelas atau fungsi. Tipe *generic* dideklarasikan di dalam kelas dan didefinisikan saat instansiasi. *Compiler* akan mengganti semua kemunculan parameter bertipe kelas tertentu dengan *upprbound* dari tipe parameter formalnya. Secara *default*, *upprbound* yang digunakan adalah tipe *Object*.

```
import java.util.*;
public class MyGenerics<E> {
    private List<E> list = new ArrayList<>();
    public int getSize() {
        return list.size();
    }
    public void insert(E item) {
        list.add(item);
    }
}
```



```

public void printAll() {
    list.forEach(System.out::println);
}
public static void main(String[] args) {
    class Building {
        String address;
        Building(String address) {
            this.address = address;
        }
        @Override
        public String toString() {
            return "address: "+address;
        }
    }
    MyGenerics<Building> building = new MyGenerics<>();
    building.insert(new Building("Jl Margahayu Raya no. 30"));
    building.insert(new Building("Jl Cisaranten no. 17"));
    building.insert(new Building("Jl Banteng no. 25"));
    building.insert(new Building("Jl Solokan Jeruk no. 14"));
    building.insert(new Building("Jl Telekomunikasi no. 6"));
    building.printAll();
}
}

```

Penggunaan wildcard `<E>` pada kelas **MyGeneric** akan membatasi kelas **MyGeneric** sesuai tipe yang diberikan pada E. Contoh pada *method* `main`, tipe E diganti dengan tipe *Building*, maka kelas *Building* akan menjadi *upprbound* dari kelas **MyGeneric**. Sehingga, objek *building* hanya akan menerima objek bertipe *Building*. Kita juga bisa menggunakan *unbounded wildcard* untuk memberikan batasan yang lebih luas kepada sebuah *generic*.

Contohnya sebagai berikut:

```

import java.util.*;

public class GenericMethod {
    public static double average(List<? extends Number> list) {
        double sum = 0;
        for(Number num:list) {
            sum += num.doubleValue();
        }
        return sum/list.size();
    }
    public static void main(String[] args) {
        List<Integer> number1 = new LinkedList<>();
        number1.add(45);
        number1.add(75);
        number1.add(35);
        System.out.println("Average number1 = "+average(number1));

        List<Double> number2 = new ArrayList<>();
        number2.add(49.23);
        number2.add(22.11);
        number2.add(32.73);
        System.out.println("Average number2 = "+average(number2));
    }
}

```

Penggunaan *unbounded wildcard* `<?>` pada deklarasi *method* `average` menyebabkan parameter pada *method* `average` dapat menerima parameter bertipe *List* dengan isi elemen apapun selama masih merupakan turunan dari kelas *Number*. Dengan adanya *unbounded wildcard*, Kita dapat memperluas definisi dari tipe *generic* yang Kita inginkan, tidak hanya satu tipe saja seperti pada contoh di atas.

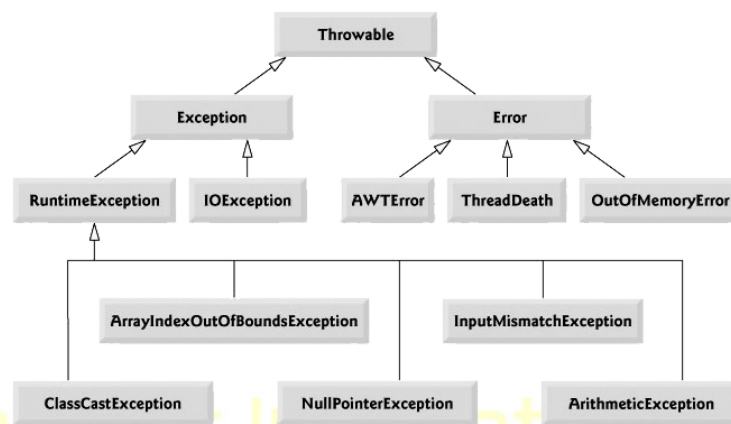
Modul 10 Exception

Tujuan Praktikum

1. Memahami konsep *Exception*

10.1 Exception

Exception adalah **kondisi abnormal / tidak wajar yang terjadi pada saat pengeksekusian** suatu perintah. Dalam java ada lima **keywords** yang digunakan untuk menangani eksepsi ini yaitu : **try, catch, finally, throw, dan throws**. Berikut adalah gambar hierarki dari tipe - tipe eksepsi.



Gambar 10-1 Hierarki dari tipe-tipe eksepsi.

Superclass tertinggi adalah **class Throwable**, tetapi kelas ini tidak akan pernah menggunakan kelas secara langsung. Dibawah **class Throwable** terdapat dua *subclass* yaitu **class Error dan class Exception**. **Class Error** merupakan tipe eksepsi yang seharusnya tidak ditangani dengan blok *try catch* dalam program, karena berhubungan dengan Java *run-time system*. Hal ini dikarenakan eksepsi yang terjadi sangat kritis. Sedangkan **class Exception** merupakan tipe eksepsi yang memang sebaiknya ditangani oleh program Anda secara langsung.

10.2 Eksepsi yang tidak dicek

Semua eksepsi yang bertipe *RuntimeException* dan turunannya tidak harus ditangani dalam program Anda.

```
class DemoEksepsi
{
    public static void main(String[] args)
    {
        int[] arr = new int[1];
        System.out.println(arr[1]);
    }
}
```

Output-nya:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 1
    at DemoEksepsi.main(DemoEksepsi.java:6)
```

10.3 Eksepsi yang dicek

Semua tipe eksepsi yang bukan turunan dari *class RuntimeException* merupakan eksepsi yang harus ditangani oleh program Anda. Java bahkan tidak mengizinkan Anda mengompilasi program yang Anda buat, jika tidak menangani eksepsi tersebut. (biasanya yang berhubungan dengan pengaksesan I/O)

10.4 Penggunaan Blok Try Catch

Untuk menangani eksepsi yang terjadi dalam program, Anda cukup meletakkan kode yang ingin Anda inspeksi terhadap kemungkinan eksepsi di dalam blok *try*, diikuti dengan blok *catch* yang menentukan jenis eksepsi apa yang ingin Anda tangani dalam blok *catch* tersebut.

Contoh program:

```
class DemoEksepsi {
    public static void main(String[] args) {
        try {
            int[] arr = new int[1];
            System.out.println(arr[1]);
            System.out.println("Baris ini tidak akan pernah dieksekusi...");
        } catch(ArithmeticException e) {
            System.out.println("Terjadi eksepsi karena indeks di luar kapasitas array");
        }
        System.out.println("Sesudah blok try catch");
    }
}
```

Output-nya:

```
Terjadi eksepsi karena indeks di luar kapasitas array
Sesudah blok try catch
```

Dapat terjadi kode yang terdapat dalam **blok try** mengakibatkan lebih dari satu jenis eksepsi. Dalam hal ini, Kita dapat menuliskan lebih dari satu blok *catch* untuk setiap blok *try*. Berikut adalah contohnya.

```
class DemoEksepsi {
    public static void main(String[] args){
        try {
            int x= args.length;
            int y= 100/x;
            int[] arr={10,11};
            y = arr[x];
            System.out.println("Tidak terjadi eksepsi");
        } catch(ArithmeticException e) {
            System.out.println("Terjadi eksepsi karena pembagian dengan nol");
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Setelah blok try catch");
        }
    }
}
```

Output program di atas tergantung jumlah parameter yang diberikan. Jika tanpa parameter, maka output-nya :

```
C:\Java_projects> java DemoEksepsi
Terjadi eksepsi karena pembagian dengan nol
Setelah blok try catch
```

Jika dieksekusi dengan sebuah parameter maka output-nya:

```
C:\Java_projects> java DemoEksepsi param1
Tidak terjadi eksepsi
Setelah blok try catch
```

Jika dieksekusi dengan dua buah parameter maka output-nya:

```
C:\Java_projects> java DemoEksepsi
Terjadi eksepsi karena indeks di luar kapasitas array
Setelah blok try catch
```

10.5 Penggunaan Throws

Penggunaan *keyword* ini berhubungan erat dengan penggunaan *exception* yang dicek oleh Java. Setiap *method* yang mungkin menyebabkan suatu *exception* dan tidak menangani *exception* tsb, dalam arti *exception* tsb akan dilempar ke luar, maka *method* tersebut harus menjelaskan kemungkinan ini agar si pemanggil *method* ini dapat mengetahui dan bersiap-siap untuk menangani *exception* yg mungkin terjadi. Ini dilakukan dengan cara menggunakan *keyword throws* pada saat pendeklarasian *method*.

Apabila Kita tidak menyertakan blok *try-catch* di dalam *method* yang mengandung kode-kode yang mungkin menimbulkan eksepsi, maka Kita harus menyertakan klausa *throws* pada saat pendeklarasian *method* yang bersangkutan. Jika tidak, maka program tidak dapat dikompilasi.

Contoh program:

```
class Coba {
    public void tampil() throws Exception {
        int x=0;
        if (x<5)
            throw new Exception("Lebih kecil 5");
    }
}

public class DemoException {
    public static void main (String args[]) {
        Coba c = new Coba();
        try {
            c.tampil();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Program Selesai");
    }
}
```

Output program :

```
Lebih kecil 5
Program Selesai
```

10.6 Pemakaian Finally

Penggunaan blok *try catch* terkadang membingungkan karena Kita tidak dapat menentukan dengan pasti alur mana yang akan dieksekusi. Apalagi penggunaan *throw* yang mengakibatkan kode setelah *throw* tidak akan dieksekusi atau justru terjadi kesalahan pada blok *catch*, menyebabkan program akan berhenti. Untuk mengatasi problem ini, Java memperkenalkan *keyword finally*. Dimana semua kode yang ada dalam blok *finally* “pasti” akan dieksekusi apapun yang terjadi di dalam blok *try catch*.

Contoh:

```
public class DemoFinally{
    public static void main(String[] args){
        int x = 3;
        int[] arr = {10,11,12};
        try {
            System.out.println(arr[x]);
            System.out.println("Tidak Terjadi Eksepsi");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Terjadi Eksepsi");
            System.out.println(arr[x-4]);
        } finally {
            System.out.println("Program Selesai");
        }
    }
}
```

Output program

```
Terjadi Eksepsi
Program Selesai
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
    at DemoFinally.main(coba2.java:13)
```

Modul 11 Graphical User Interface (GUI)

Tujuan Praktikum

1. Mengetahui komponen-komponen dasar dalam pembuatan *user interface* dalam Java.
2. Mengetahui konsep *Graphic User Interface* dalam Java dan dapat mengimplementasikannya dalam sebuah program sederhana.

11.1 AWT and Swing

AWT (*Abstract Windowing Toolkit*) dan *Swing* pada dasarnya adalah sama untuk membangun GUI pada Java. AWT terdapat dalam *package* `java.awt`. *Package* ini berisi komponen-komponen GUI yang bersifat *platform oriented* atau tergantung pada suatu *platform system operation*. *Swing* ada pada *package* `javax.swing` dan bersifat *lightweight*, yaitu dapat diaplikasikan untuk semua platform (multiplatform). Dalam perkembangannya *Swing* lebih mendominasi dari pada Awt. Hal ini dapat dilihat banyaknya komponen *Swing* yang tersedia dibandingkan komponen AWT.

Beberapa fasilitas yang disediakan oleh kedua kelas tersebut adalah sebagai berikut:

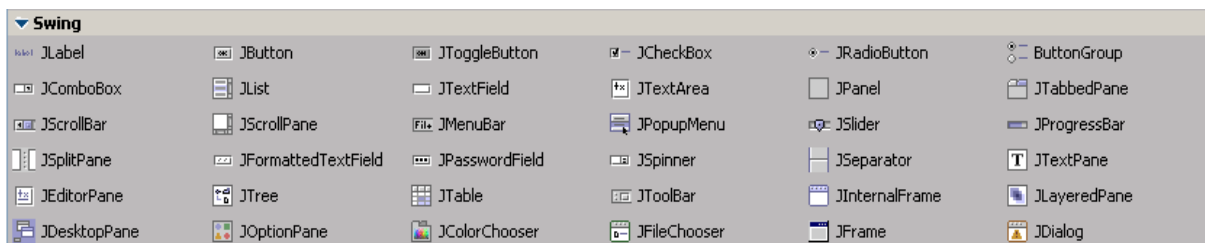
- 1) Pengaturan tata letak (*layout management*) komponen dalam suatu *container*.
- 2) Mendukung *event handling*, yaitu mekanisme pendeteksian *event* dan penentuan *respons* yang akan diberikan ketika pengguna (*user*) mengakses komponen tersebut.
- 3) Manipulasi grafis komponen seperti *font*, warna, *icon*, dll.

11.2 AWT components

- 1) Label
- 2) Checkbox
- 3) Choice
- 4) List
- 5) MenuItem
- 6) TextField
- 7) ScrollBar
- 8) TextArea
- 9) ScrollPanel
- 10) Canvas

11.3 Swing components

Komponen *Swing* hampir sama dengan yang ada di Awt, namun ada penambahan *fiture* pada *Swing*(*other*) .



Gambar 11-1 Komponen *Swing*.

11.4 Komponen utama dalam GUI

- 1) *Containers*: tempat dimana *componen* lain bisa di tempatkan didalamnya, contohnya adalah panel.
- 2) *Canvases*. Digunakan untuk menampilkan *image* atau pembuatan program yang berbasis grafik. Kita bisa menggambar titik, lingkaran dan sebagainya.

- 3) *UIcomponents*. contohnya *buttons, lists, simple pop-up menus, check boxes, text fields*, dan elemen khusus lain untuk *user interface*.
- 4) *Window construction components*. Contohnya *windows, frames, menu bars*, dan *dialog boxes*.

11.5 The Basic User Interface Components with Swing

Secara umum terdapat 5 bagian Swing akan sering digunakan :

1) Top-Level Container

Container dasar dimana komponen lain diletakkan.

Ex: *Frame (JFrame)*, *Dialog (JDialog)* & *Applet (JApplet)*

2) Intermediate Container

Container perantara dimana komponen lain diletakkan

Ex: *JPanel*, dimana umumnya hanya digunakan sebagai tempat untuk meletakkan/mengelompokkan komponen-komponen yang digunakan, baik *container* atau berupa *atomic component*. Dan digunakan juga sebagai *scroll pane* (*JScrollPane* & *JTabbedPane*).

3) Atomic Component

Komponen yang memiliki fungsi spesifik, dimana umumnya **user langsung berinteraksi** dengan komponen ini

Ex: *JButton*, *JLabel*, *TextField*, *TextArea*.

11.6 Top Level Container

- 1) *Frame (JFrame)* : *Constructor* yang dapat digunakan untuk membuat *frame* adalah *JFrame()* dan *JFrame(String title)*. Contoh penggunaannya adalah sbb:

```
import javax.swing.*;
public class FrameDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Frame Demo");
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
        frame.setSize(400,150);
        frame.show();
    }
}
```

Class *JFrame* mendefinisikan 4 aktifitas yaitu:

a. DO_NOTHING_ON_CLOSE

Tidak ada aktivitas apapun yang dilakukan secara otomatis jika Kita menutup *frame* tsb. Biasanya digunakan jika Kita ingin menangani sendiri aktivitas tsb

b. HIDE_ON_CLOSE

Merupakan aktivitas default, dimana *frame* hanya disembunyikan atau tidak ditampilkan ke layar, namun secara fisik *frame* ini masih ada di memori sehingga jika diinginkan dapat ditampilkan kembali.

c. **DISPOSE_ON_CLOSE**

Menghapus tampilan *frame* dari layar, menghapusnya dari memori & membebaskan *resource* yang dipakai

d. **EXIT_ON_CLOSE**

Menghentikan eksekusi program. Cocok digunakan untuk *frame* utama, dimana jika *frame* tsb ditutup mengakibatkan eksekusi program berhenti

- 2) Dialog (JDialog): Perbedaan utama *Frame* dengan Dialog adalah: Dialog tidak berdiri sendiri (biasa dibuat bersama dengan *frame* sebagai parentnya sehingga Dialog akan dihapus dari memori jika *Frame* parentnya juga dihapus). Dialog bersifat modal (memblok semua input terhadap *parent*-nya sampai Dialog tersebut ditutup).

Contoh penggunaanya adalah sbb:

```
import javax.swing.*;
public class DialogDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Contoh Frame"); // frame sbg parent
        bagi
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.show();
            JOptionPane.showConfirmDialog(frame,
                "Contoh dialog konfirmasi ...", //Informasi dialog
                "Judul Dialog", //judul dialog
                JOptionPane.OK_CANCEL_OPTION, //Jenis Tombol
                JOptionPane.QUESTION_MESSAGE); //Icon
    }
}
```

11.7 Pengaturan Layout

- 1) Pengaturan dengan BorderLayout
- 2) Pengaturan dengan BoxLayout
- 3) Pengaturan dengan CardLayout
- 4) Pengaturan dengan FlowLayout
- 5) Pengaturan dengan GridLayout

11.7.1 Pengaturan dengan BorderLayout

BorderLayout adalah kelas yang menempatkan komponen dengan pendekatan arah mata angin. Pada program dibawah ini kelima tombol diletakkan pada posisi yang berbeda-beda, ditentukan konstanta: *NORTH*, *EAST*, *SOUTH*, *WEST*, *CENTER*.



Gambar 11-2 BorderLayout.

Source Code-nya:

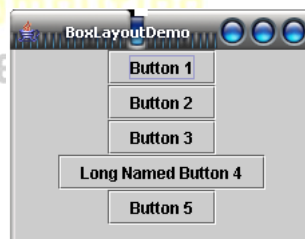
```
// file BorderLayoutDemo.java
import javax.swing.*;
import java.awt.*;

public class BorderLayoutDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame ("Contoh Border Layout");
        BorderLayout layout = new BorderLayout(1,1);
        frame.getContentPane().setLayout(layout);

        JButton btnNorth = new JButton("Posisi NORTH");
        JButton btnSouth = new JButton("Posisi SOUTH");
        JButton btnEast = new JButton("Posisi EAST");
        JButton btnWest = new JButton("Posisi WEST");
        JButton btnCenter = new JButton("Posisi CENTER");
        frame.getContentPane().add(btnNorth, BorderLayout.NORTH);
        frame.getContentPane().add(btnSouth, BorderLayout.SOUTH);
        frame.getContentPane().add(btnEast, BorderLayout.EAST);
        frame.getContentPane().add(btnWest, BorderLayout.WEST);
        frame.getContentPane().add(btnCenter, BorderLayout.CENTER);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.show();
    }
}
```

11.7.2 Pengaturan dengan BoxLayout

BoxLayout meletakkan komponen- komponen dalam satu baris atau kolom saja.



Gambar 11-3 BoxLayout.

Source Code-nya:

```
// file BoxLayoutDemo.java
import javax.swing.*;
import java.awt.*;

public class BoxLayoutDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame ("BoxLayoutDemo");
        Box comp = new Box(BoxLayout.Y_AXIS); //X_AXIS

        JButton btn1 = new JButton("Button 1");
        JButton btn2 = new JButton("Button 2");
        JButton btn3 = new JButton("Button 3");
        JButton btn4 = new JButton("Long Named Button 4");
        JButton btn5 = new JButton("Button 5");

        btn1.setAlignmentX(Component.CENTER_ALIGNMENT);
        btn2.setAlignmentX(Component.CENTER_ALIGNMENT);
        btn3.setAlignmentX(Component.CENTER_ALIGNMENT);
        btn4.setAlignmentX(Component.CENTER_ALIGNMENT);
        btn5.setAlignmentX(Component.CENTER_ALIGNMENT);

        comp.add(btn1);
        comp.add(btn2);
        comp.add(btn3);
        comp.add(btn4);
        comp.add(btn5);

        frame.getContentPane().add(comp);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.show();
    }
}
```

11.7.3 Pengaturan dengan CardLayout

CardLayout berguna untuk **menampilkan objek Container** seperti tumpukan kartu. Oleh karena itu, **hanya satu Container yang tertampil** untuk setiap waktu.



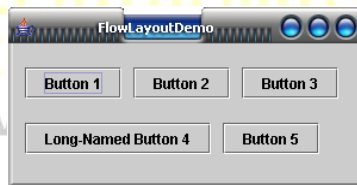
Gambar 11-4 CardLayout.

Source code-nya:

```
import javax.swing.*;
import java.awt.*;
public class CardLayoutDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame ("CardLayoutDemo");
        JPanel panel1 = new JPanel();
        JPanel panel2 = new JPanel();
        JButton btn1 = new JButton("Tombol 1");
        JButton btn2 = new JButton("Tombol 2");
        JTextField text1 = new JTextField(20);
        panel1.add(btn1);
        panel1.add(btn2);
        panel2.add(text1);
        JTabbedPane tab = new JTabbedPane();
        tab.add(panel1,"Tab dengan Button");
        tab.add(panel2,"Tab dengan TextField");
        frame.getContentPane().add(tab,BorderLayout.NORTH);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300,150);
        frame.show();
    }
}
```

11.7.4 Pengaturan dengan FlowLayout

FlowLayout melakukan pengaturan komponen – komponen dalam objek *Container* dengan urutan dari kiri ke kanan dan dari atas ke bawah. Pengaturan rata kiri, rata kanan, atau rata tengah bisa dilakukan dengan melibatkan konstanta *LEFT*, *RIGHT*, *CENTER*.



Gambar 11-5 FlowLayout.

Source Code-nya:

```
import javax.swing.*;
import java.awt.*;
public class FlowLayoutDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame ("FlowLayoutDemo");
        FlowLayout layout = new FlowLayout(FlowLayout.LEFT);
        layout.setVgap(20);    //jarak vertikal antar komponen
        layout.setHgap(10);    //jarak horisontal antar komponen
        frame.getContentPane().setLayout(layout);
        JButton btn1 = new JButton("Button 1");
        JButton btn2 = new JButton("Button 2");
        JButton btn3 = new JButton("Button 3");
        JButton btn4 = new JButton("Long-Named Button 4");
        JButton btn5 = new JButton("Button 5");
        frame.getContentPane().add(btn1);
        frame.getContentPane().add(btn2);
        frame.getContentPane().add(btn3);
        frame.getContentPane().add(btn4);
        frame.getContentPane().add(btn5);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300,150);
        frame.show();
    }
}
```

11.7.5 Pengaturan dengan GridLayout

GridLayout merupakan manajer tata letak komponen yang menggunakan bentuk *grid* dengan ukuran yang sama untuk setiap komponen.



Gambar 11-6 GridLayout.

Source Code-nya:

```
import javax.swing.*;
import java.awt.*;

public class GridLayoutDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame ("GridLayoutDemo");
        GridLayout layout = new GridLayout(3,2);
        layout.setHgap(5);
        layout.setVgap(10);
        frame.getContentPane().setLayout(layout);
        JButton btn1 = new JButton("Button 1");
        JButton btn2 = new JButton("Button 2");
        JButton btn3 = new JButton("Button 3");
        JButton btn4 = new JButton("Long-Named Button 4");
        JButton btn5 = new JButton("Button 5");
        frame.getContentPane().add(btn1);
        frame.getContentPane().add(btn2);
        frame.getContentPane().add(btn3);
        frame.getContentPane().add(btn4);
        frame.getContentPane().add(btn5);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.show();
    }
}
```

11.8 Event Handling

Untuk mendeteksi atas apa yang dilakukan oleh user, diperlukan **penanganan khusus** terhadap **event** (peristiwa yang di-stimulasikan/di-trigger) yang diperlukan oleh *user* terhadap **komponen GUI** tertentu. Penanganan ini disebut **Event Handling Component**. *Event Handling Component* dibagi atas 2 bagian, yaitu **Event Listener** dan **Event Handler**. Untuk mengenal 2 *method Event* tersebut dapat diilustrasikan sbb:

Bila mengklik suatu *Object button*, maka tercipta *event*. *Event* ini ditangkap oleh *Event Listener* dan secara otomatis akan diberi ID (identitas) seperti *button1*, *button2*, *textfield1*, dll atau Kita dapat memberikan ID atas komponen tersebut dengan memberi nama suatu *Object*. Sehingga Java mengenal *event* mana yang ditangkap.

Selanjutnya, tentukan **Event Handler** dari komponen tersebut, berupa **blok program/statement** yang **memproses** bila terjadi suatu *event* yang ditangkap oleh **Event Listener**. Berikut merupakan contoh program dari *event handling* yang menggunakan *JTable*, *JButton*, *JRadioButton*, *JCheckBox*, dan *JComboBox*.

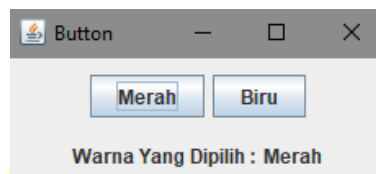
Berikut jenis-jenis Event Listener yang ada pada Java:

Tabel 12-1 Beberapa Jenis Event Listener

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

11.8.1 Button

Setiap *Button* yang dirancang harus dibuat *event* agar bisa melakukan aksi, dalam contoh berikut tombol merah dan biru akan mengubah label dari warna yang dipilih



Gambar 11-7 Contoh Penggunaan Button.

Source Code-nya:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class button{
    private final JLabel label, labelWarna;
    private final JFrame frame;
    private final JButton buttonMerah,buttonBiru;

    public button(){
        frame          = new JFrame("Button");
        label           = new JLabel("Warna Yang Dipilih :");
        labelWarna      = new JLabel();
        buttonMerah     = new JButton("Merah");
        buttonBiru      = new JButton("Biru");
    }

    private class merahHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            labelWarna.setText("Merah");
        }
    }

    private class biruHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            labelWarna.setText("Biru");
        }
    }

    public void launch(){
        FlowLayout layout = new FlowLayout(FlowLayout.CENTER);
        frame.getContentPane().setLayout(layout);
        buttonMerah.addActionListener(new merahHandler());
        buttonBiru.addActionListener(new biruHandler());
        JPanel panel1 = new JPanel();
        JPanel panel2 = new JPanel();
        panel1.add(buttonMerah);
        panel1.add(buttonBiru);
        panel2.add(label);
        panel2.add(labelWarna);
        frame.getContentPane().add(panel1);
        frame.getContentPane().add(panel2);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(250,150);
        frame.show();
    }

    public static void main(String[] args) {
        button gui = new button();
        gui.launch();
    }
}
```

Contoh diatas, kelas **merahHandler** menangani terhadap *event* dengan mengimplementasi dari *Event Listener* yaitu *Action Listener*. *Action Listener* akan menangkap *event* dari **buttonMerah** saat di tekan, begitu pula dengan *event handler* yang lain.

11.8.2 ComboBox

Penggunaan *ComboBox* dilakukan dengan menambah item *list* pada *ComboBox* saat instansiasi Objek, berikut contoh penggunaan *ComboBox*



Gambar 11-8 Contoh Penggunaan ComboBox.

Source Code-nya:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class comboBox{
    private final JLabel label;
    private final JFrame frame;
    private final JButton button;
    private final JComboBox cB;

    public comboBox(){
        frame = new JFrame("ComboBox");
        label = new JLabel("Fakultas : ");
        button = new JButton("Tampilkan");
        String item[] = {"FIF", "FKB", "FIT", "FEB", "FTE", "FRI", "FIK"};
        cB = new JComboBox(item);
    }

    private class tampilHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(frame, cB.getSelectedItem(),
                "Fakultas Yang Dipilih", JOptionPane.PLAIN_MESSAGE);
        }
    }

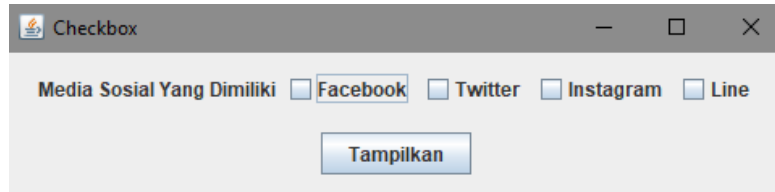
    public void launch(){
        FlowLayout layout = new FlowLayout(FlowLayout.CENTER);
        frame.getContentPane().setLayout(layout);
        button.addActionListener(new tampilHandler());
        JPanel panel1 = new JPanel();
        JPanel panel2 = new JPanel();
        panel1.add(label);
        panel1.add(cB);
        panel2.add(button);
        frame.getContentPane().add(panel1);
        frame.getContentPane().add(panel2);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300,100);
        frame.show();
    }

    public static void main(String[] args) {
        comboBox gui = new comboBox();
        gui.launch();
    }
}
```

Contoh diatas, kelas **tampilHandler** menangani terhadap *event* dengan mengimplementasi dari *Event Listener* yaitu *Action Listener*. Lalu blok program/statement pada **tampilHandler** akan memproses bila terjadi suatu *event* yang ditangkap oleh *Event Listener*. Pada contoh diatas *event* yang dilakukan yaitu pada *button* dengan menampilkan data sesuai yang ada pada **comboBox**.

11.8.3 CheckBox

Penggunaan *CheckBox* dapat menggunakan method *actionCommand* atau method *getText* saat ingin mendapatkan *value CheckBox*-nya, pada contoh berikut digunakan method *getText* pada penggunaan *CheckBox*



Gambar 11-9 Contoh Penggunaan CheckBox.

Source Code-nya:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class checkbox{
    private final JLabel label;
    private final JFrame frame;
    private final JButton button;
    private final JCheckBox c1,c2,c3,c4;

    public checkbox(){
        frame = new JFrame("Checkbox");
        label = new JLabel("Media Sosial Yang Dimiliki");
        button = new JButton("Tampilkan");
        c1 = new JCheckBox("Facebook");
        c2 = new JCheckBox("Twitter");
        c3 = new JCheckBox("Instagram");
        c4 = new JCheckBox("Line");
    }

    private class tampilHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String cs1,cs2,cs3,cs4;
            cs1 = c1.getText();
            cs2 = c2.getText();
            cs3 = c3.getText();
            cs4 = c4.getText();
            JOptionPane.showMessageDialog(frame,cs1 + ", " + cs2 + ", "
+
            cs3 + ", " + cs4,"Media Sosial Yang Dipilih",
            JOptionPane.PLAIN_MESSAGE);
        }
    }

    public void launch(){
        FlowLayout layout = new FlowLayout(FlowLayout.CENTER);
        frame.getContentPane().setLayout(layout);
        button.addActionListener(new tampilHandler());
        JPanel panel1 = new JPanel();
        JPanel panel2 = new JPanel();
        panel1.add(label);
        panel1.add(c1);
        panel1.add(c2);
        panel1.add(c3);
        panel1.add(c4);
        panel2.add(button);
        frame.getContentPane().add(panel1);
        frame.getContentPane().add(panel2);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

```

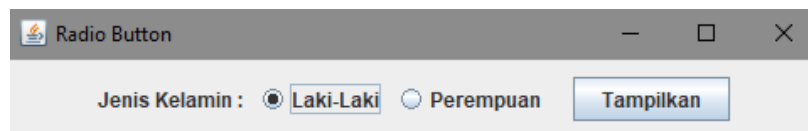
        frame.setSize(500,150);
        frame.show();
    }
    public static void main(String[] args) {
        checkbox gui = new checkbox();
        gui.launch();
    }
}

```

Contoh diatas, kelas **tampilHandler** menangani terhadap event dengan mengimplementasi dari *Event Listener* yaitu *ActionListener*. Lalu blok program/statement pada **tampilHandler** akan memproses bila terjadi suatu event yang ditangkap oleh Event Listener. Pada contoh diatas event yang dilakukan yaitu pada *button* dengan menampilkan data sesuai yang ada pada **CheckBox**.

11.8.4 RadioButton

Pada penggunaan *RadioButton* perlu dilakukan penambahan *actionCommand* dari setiap *RadioButton* dan dilakukan *group button* agar hanya salah satu *RadioButton* yang aktif, dalam contoh berikut *user* hanya bisa memilih jenis kelamin Laki-laki atau Perempuan



Gambar 11-10 Contoh Penggunaan RadioButton.

Source Code-nya:

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class radioButton{
    private final JLabel label;
    private final JFrame frame;
    private final JButton button;
    private final JRadioButton r1,r2;
    private final ButtonGroup group;

    public radioButton(){
        frame = new JFrame("Radio Button");
        label = new JLabel("Jenis Kelamin : ");
        button = new JButton("Tampilkan");
        r1 = new JRadioButton("Laki-Laki");
        r1.setActionCommand("Laki-Laki");
        r2 = new JRadioButton("Perempuan");
        r2.setActionCommand("Perempuan");
        group = new ButtonGroup();

    }

    private class tampilHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(frame,
                group.getSelection().getActionCommand(),
                "Jenis Kelamin Yang Dipilih",
                JOptionPane.PLAIN_MESSAGE);
        }
    }

    public void launch(){
        FlowLayout layout = new FlowLayout(FlowLayout.CENTER);
        frame.getContentPane().setLayout(layout);
    }
}

```



```

        button.addActionListener(new tampilHandler());
        JPanel panel1 = new JPanel();
        JPanel panel2 = new JPanel();
        panel1.add(label);
        panel1.add(r1);
        panel1.add(r2);
        group.add(r1);
        group.add(r2);
        panel2.add(button);

        frame.getContentPane().add(panel1);
        frame.getContentPane().add(panel2);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(500,150);
        frame.show();
    }

    public static void main(String[] args) {
        radioButton gui = new radioButton();
        gui.launch();
    }
}

```

Contoh diatas, kelas **tampilHandler** menangani terhadap event dengan mengimplementasi dari *Event Listener* yaitu *ActionListener*. Lalu blok program/statement pada **tampilHandler** akan memproses bila terjadi suatu event yang ditangkap oleh *Event Listener*. Pada contoh diatas event yang dilakukan yaitu pada button dengan menampilkan data sesuai yang ada pada **RadioButton**.

11.8.5 Table

Pada penggunaan tabel bisa menggunakan tabel statis maupun dinamis, pada contoh berikut digunakan tabel dinamis untuk menginput nama dan jenis kelamin

Nama	Jenis Kelamin
Hermawan	Laki-Laki
Suciati	Perempuan

Gambar 11-11 Contoh Penggunaan Table.

Source Code-nya:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.DefaultTableModel;

public class contohTabel{
    private final JLabel label1, label2;
    private final JTextField text1, text2;
    private final JFrame frame;
    private final JButton button1, button2, button3;
    private final JTable tabel;
    private final DefaultTableModel model;
    private final JScrollPane scrollPane;
    public contohTabel(){
        frame = new JFrame("Tabel");
        label1 = new JLabel("Nama : ");
        label2 = new JLabel("Jenis Kelamin : ");
        text1 = new JTextField(15);
        text2 = new JTextField(15);
        button1 = new JButton("Simpan");
        button2 = new JButton("Hapus");
        button3 = new JButton("Clear");
        tabel = new JTable();
        model = new DefaultTableModel();
        scrollPane = new JScrollPane(tabel);
    }

    private class simpanHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            Object[] obj = new Object[2];
            obj[0] = text1.getText();
            obj[1] = text2.getText();
            model.addRow(obj);
            text1.setText("");
            text2.setText("");
        }
    }

    private class clearHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            model.getDataVector().removeAllElements();
            model.fireTableDataChanged();
        }
    }

    private class hapusHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            try{
                model.removeRow(tabel.getSelectedRow());
            }catch (ArrayIndexOutOfBoundsException ex){
                JOptionPane.showMessageDialog(frame,
                    "Tidak Ada Baris Tabel Yang Dipilih",
                    "Peringatan", JOptionPane.PLAIN_MESSAGE);
            }
        }
    }

    public void launch(){
        FlowLayout layout = new FlowLayout(FlowLayout.CENTER);
        frame.getContentPane().setLayout(layout);
        button1.addActionListener(new simpanHandler());
        button2.addActionListener(new hapusHandler());
        button3.addActionListener(new clearHandler());
        JPanel panel1 = new JPanel();
        JPanel panel2 = new JPanel();
    }
}
```

```

        JPanel panel3 = new JPanel();
        JPanel panel4 = new JPanel();
        panel1.add(label1);
        panel1.add(text1);
        panel2.add(label2);
        panel2.add(text2);
        panel3.add(button1);
        panel3.add(button2);
        panel3.add(button3);
        tabel.setModel(model);
        model.addColumn("Nama");
        model.addColumn("Jenis Kelamin");
        tabel.getColumnModel().getColumn(0).setPreferredWidth(200);
        tabel.getColumnModel().getColumn(1).setPreferredWidth(125);
        scrollPane.setPreferredSize(new Dimension(450,100));
        panel4.add(scrollPane);
        frame.getContentPane().add(panel1);
        frame.getContentPane().add(panel2);
        frame.getContentPane().add(panel3);
        frame.getContentPane().add(panel4);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(500,275);
        frame.show();
    }
    public static void main(String[] args) {
        contohTabel gui = new contohTabel();
        gui.launch();
    }
}

```

Contoh diatas, *Event Handler* menangani terhadap *event* dengan mengimplementasi dari *Event Listener* yaitu *Action Listener*. Lalu blok program/statement pada **simpanHandler**, **clearHandler**, dan **hapusHandler** akan memproses bila terjadi suatu event yang ditangkap oleh *Event Listener*. Pada contoh diatas *Event Handler* akan mengubah data **Table** sesuai statement program pada kelas *Event Handler*.

Modul 12 Input Output dan File

Tujuan Praktikum

1. Memahami konsep *Input/Output* yang ada dalam java.
2. Mengimplementasikan operasi file dalam Java.

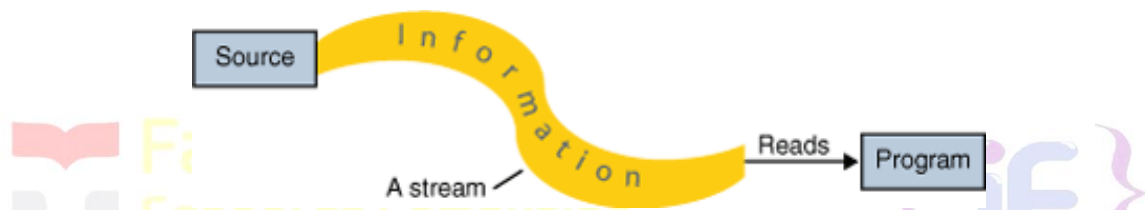
12.1 Input Output

12.1.1 Java I/O

Java io adalah paket untuk **menangani pembacaan dan penulisan informasi** untuk pemrograman java. Dalam Java I/O terbagi dalam 2 tipe. **Byte Stream** dan **Character Stream**. Sebelum lebih jauh lagi Kita membahasnya akan di jelaskan terlebih dahulu konsep I/O dan *Stream*.

I/O adalah input dan output. Yang digunakan **untuk mengambil informasi data** dari perangkat keras untuk metoda **input**, sedangkan **output untuk mengirim informasi** data ke perangkat keras.

Stream adalah **aliran proses informasi data yang direpresentasikan secara abstrak** untuk sebuah input atau output. Kita dapat menuliskan data ke sebuah *Stream* dan membaca data dari sebuah *Stream*.



Gambar 12-1 Alur membaca Stream.

Dari Gambar 8-2 sebuah program **membuka Stream** dari sebuah **sumber informasi (source)** seperti *file, memory, socket*. Kemudian membaca informasi secara **sekuensial**.

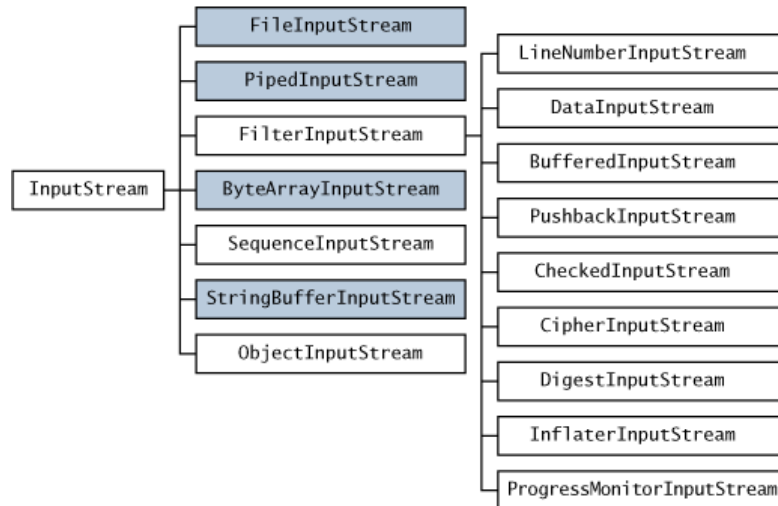


Gambar 12-2 Alur menulis Stream.

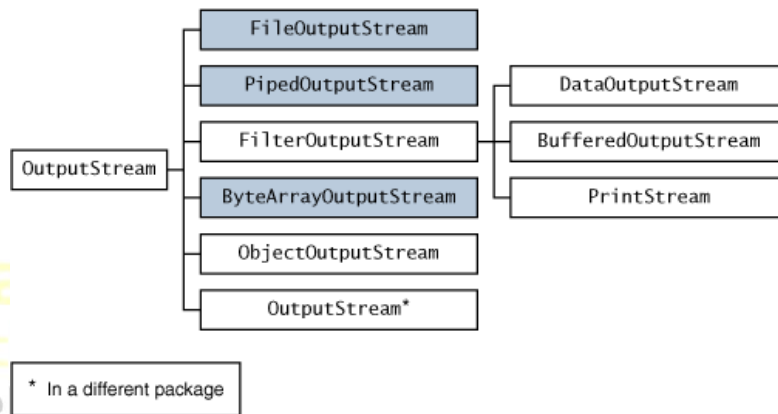
Dari Gambar 8-3 sebuah program dapat **mengirim informasi** dengan **membuka Stream** ke sebuah **tujuan informasi (destination)**. Kemudian menuliskan informasi ke tujuan secara **sekuensial**.

12.1.2 Byte Stream

Byte Stream atau binari *Stream* berisikan **binari data**. Pada java *Byte Stream* mempunyai 2 buah *superclass* yaitu **InputStream** dan **OutputStream** yang merupakan kelas **abstract**.



Gambar 12-3 Hirarki kelas dari class InputStream.



Gambar 12-4 Hirarki kelas dari class OutputStream.

Gambar di atas merupakan hirarki kelas dari *class InputStream* dan *class OutputStream*.

- 1) Data *InputStream* mempunyai **1 konstruktor** dan **17 method** yang ada dalam kelasnya **selebihnya turunan** dari kelas **parent** nya. Berikut ini akan dijelaskan konstruktor dan *Method* yang sering digunakan.

- a. **DataInputStream(InputStream in)**

Untuk membuat objek *DataInputStream* dengan spesifikasi **InputStream** tertentu.

- b. **xxx readXxx()**

xxx disini dapat diganti dengan tipe data primitif seperti int, float, boolean, Byte, char, dan sebagainya. Digunakan untuk membaca dari *Stream* tipe data tertentu secara langsung.

- 2) Data *OutputStream* mempunyai **1 konstruktor** dan **15 method** yang ada dalam kelasnya **selebihnya turunan** dari kelas **parent** nya. Berikut ini akan dijelaskan konstruktor dan *method* yang sering digunakan.

- a. **DataOutputStream(OutputStream out)**

Untuk membuat objek **DataOutputStream** dengan spesifikasi **OutputStream** tertentu.

b. **void writeXxx(xxx v)**

xxx disini dapat diganti dengan **tipe data primitif** seperti int, float, Byte, boolean, Byte, char, dan sebagainya.

```
import java.io.*;
public class DemoStream {
    public static void main(String[] args) {
        Byte[] data = new Byte[10];
        int panjang=0;
        System.out.print("Masukkan data      : ");
        try {
            panjang=System.in.read(data);
            System.out.print("Yang Anda ketik  : ");
            System.out.write(data);
            System.out.println("Panjang Karakter : "+panjang);
        }
        catch (IOException e) {
            System.out.print("Terjadi Exception");
        }
    }
}
```

12.2 Operasi File

File digunakan sebagai media penyimpan, untuk mengakses *file* Kita harus menspesifikasikan dimana *file* yang akan Kita akses, atau membuat *file* baru yang akan disimpan. Dalam java Kita dapat mengakses *file* menulis dan membaca dengan *character Stream* atau dengan *Byte Stream*, dan dapat juga Kita hanya menciptakan sebuah *file*. Untuk menciptakan sebuah *file* dengan mengakses class *java.io.File* dan menciptakan objek dari kelas tersebut Kita sudah dapat menciptakan *file*. Tanpa harus menangkap *error io*. Berbeda dengan menciptakan *file* yang langsung diakses oleh *Stream*. *File* tersebut harus dapat menangkap *error io* ketika penciptaan objek kelas *file*.

12.2.1 java.io.File

Terdapat 4 atribut, 4 konstruktor dan 39 *method* yang ada didalam kelas untuk menspesifikasikan file yang dibuat.

File(),File(String path),File(String dir,String nm)

Konstruktor diatas adalah yang sering digunakan yaitu membuat objek *file* kemudian digunakan dengan pengesetan *method*-nya, atau menginstan langsung dengan nama *file* beserta *path*-nya.

boolean createNewFile(), boolean delete(),boolean exists()

Method-method diatas untuk mengeset dengan pengecekan, untuk **createNewFile()** digunakan untuk menciptakan *file* kemudian mengembalikan nilai true jika file dibuat.

12.2.2 java.io.FileOutputStream

Terdapat 5 konstruktor dan 7 *Method* untuk membuat *file* yang akan diakses menggunakan *Byte Stream*.

FileOutputStream(File of), FileOutputStream(File of,boolean append).

Digunakan untuk penciptaan objek *file* yang akan diakses dengan *Byte Stream*, dan untuk variabel *append* digunakan untuk apakah isi *file* akan dilanjutkan ke akhir dari isi *file*.

12.2.3 java.io.FileInputStream

Terdapat **3 konstruktor** dan **9 method** yang ada. Digunakan untuk **mengambil file** yang telah dideskripsikan untuk dibaca dengan **Byte Stream**.

FileInputStream(File of), FileInputStream(String nama).

Digunakan untuk **mengambil file** untuk dibaca secara **Byte Stream**, bisa **memasukan deskripsi file** yang telah ada dengan **String**, atau dengan **file** yang telah **diinstan** dengan jelas.

Contoh penggunaannya sebagai berikut:

```
import java.io.*;

public class CopyFile {
    public static void main(String[] args) throws IOException {
        String isi="Ini adalah isi file-nya";
        /* buat objek File*/
        File FAwal = new File("Fawal.txt");
        File FAakhir = new File("Fakhir.txt");

        /* Menuliskan isi ke dalam file */
        FileOutputStream fos=new FileOutputStream(FAwal);
        DataOutputStream dos=new DataOutputStream(fos);
        dos.writeBytes(isi);

        /* Membaca isi file awal*/
        FileInputStream fis=new FileInputStream(FAwal);
        DataInputStream dis=new DataInputStream(fis);
        String isiFawal= dis.readLine();
        System.out.println("isi file awal: "+isiFawal);

        /* Mengcopy isi file awal yang dibaca ke file akhir */
        FileOutputStream fos2=new FileOutputStream(FAakhir);
        DataOutputStream dos2=new DataOutputStream(fos2);
        dos2.writeBytes(isiFawal);
        System.out.println("isi file akhir: "+isiFawal);
    }
}
```

12.3 Object Serialization

Pada pemrograman io dan *socket*, yang dikirimkan hanya data *Stream* ke *file* atau ke jaringan. Lalu bagaimana jika yang dikirimkan suatu Objek? Diperlukan suatu mekanisme untuk membuat sebuah objek dapat dikirimkan seperti mengirimkan data. Dan ini yang lebih dikenal dengan Serialisasi Objek (*Object Serialization*).

Serialisasi objek perluasan dari **inti class java io** yang digunakan untuk objek. Serialisasi objek bisa digunakan untuk pengkodean (*encoding*) untuk objek dan membuat objek tersebut dapat diraih atau digunakan, melalui *bit-bit Stream*, kemudian dapat digunakan untuk penyusunan kembali objek dari *bit-bit Stream* yang dikodekan, dan saling melengkapi. Serialisasi merupakan mekanisme yang ringan dan kuat untuk komunikasi dengan sockets atau RMI(*Remote Method Invocation*). Selain untuk komunikasi dengan *sockets* teknik ini dapat digunakan juga untuk menyimpan keadaan suatu status dari suatu objek ke dalam file, seperti yang sudah dijelaskan di pendahuluan.

Jika ingin membuat sebuah objek yang dapat diserialisasikan, maka Kita harus mengimplementasikan salah satu *interface* **java.io.Serializable** atau **java.io.Externalizable** pada *class* yang ingin dibuat objek yang dapat diserialisasikan. Untuk lebih jelas tentang serialisasi objek dibawah ini akan diberikan sedikit penjelasan yang terkait dengan objek serialisasi.

12.3.1 Interface java.io.Serializable

Interface serializable harus diimplementasikan jika ingin membuat objek yang dapat diserialisasi. Implementasinya tergolong sederhana karena tidak terdapat *method* yang harus didefinisikan untuk di *override*.

Tujuan mengimplementasikan *interface serializable* adalah untuk memberitahukan kepada JVM (*Java Virtual Machine*), bahwa objek yang menerapkan *serializable* merupakan objek yang dapat diserialisasikan.

12.3.2 Class java.io.ObjectOutputStream

Class ObjectOutputStream digunakan untuk mengirimkan objek menjadi *Stream* yang lalu dapat dikirimkan ke file atau ke socket (jaringan). Class *ObjectOutputStream* mempunyai 2 constructor dan 31 *method* untuk versi JDK 1.5.

Adapun *Method* dan constructor yang sering digunakan adalah:

- 1) `ObjectOutputStream(OutputStream out)`

Membuat *ObjectOutputStream* yang akan menuliskan ke spesifik *OutputStream* yang dikehendaki.

- 2) `void writeObject(Object obj)`

Menuliskan objek yang akan dikirimkan ke *ObjectOutputStream*.

12.3.3 Class java.io.ObjectInputStream

Class ObjectInputStream adalah kelas untuk mengambil objek dari *Stream* yang dikirimkan melalui file atau socket (jaringan). Class *ObjectInputStream* mempunyai 2 Constructors, 31 *methods* untuk versi JDK 1.5.

Method dan *constructor* yang sering digunakan adalah

- 1) `ObjectInputStream(InputStream in)`

Membuat *ObjectInputStream* yang mengambil spesifik *Stream* dari *InputStream* yang dikehendaki.

- 2) `Object readObject()`

Membaca objek yang telah didefinisikan.

Contoh penerapan Serialisasi objek:

```
import java.io.*;

public class BarangSer implements Serializable{

    private String nama;
    private int jumlah;
    public BarangSer(String nm,int jml) {
        nama = nm;
        jumlah=jml;
    }

    public void tampil(){
        System.out.println("nama barang : "+nama);
        System.out.println("jumlah barang : "+jumlah);
    }

    public void simpanObject(BarangSer ob){
        try {
            FileOutputStream fos=new FileOutputStream("dtBrg.txt");
            ObjectOutputStream oos=new ObjectOutputStream(fos);
            oos.writeObject(ob);
            oos.flush();
        } catch(IOException ioe){
            System.err.println("error "+ioe);
        }
    }

    public void bacaObject(BarangSer obb){
        try {
            FileInputStream fis =new FileInputStream("dtBrg.txt");
            ObjectInputStream ois=new ObjectInputStream(fis);
            While ((obb = (BarangSer)ois.readObject()) != null)
                obb.tampil();
        } catch(IOException ioe){
            System.exit(1);
        } catch(Exception e){
            System.exit(1);
        }
    }

    public static void main(String[] args) {
        BarangSer a1=new BarangSer("Baju",5);
        a1.simpanObject(a1);
        a1.bacaObject(a1);
    }
}
```

Output program:

```
nama barang : Baju
jumlah barang : 5
```

Modul 13 JDBC (Java Database Connectivity)

Tujuan Praktikum

1. Mengimplementasikan JDBC ke dalam suatu program java

Java juga menyediakan fungsi untuk menghubungkan suatu aplikasi yang dibangun dengan menggunakan bahasa pemrograman java. Untuk menghubungkannya dengan menggunakan JDBC. Dengan JDBC, programmer dapat **menggunakan SQL statement untuk membentuk, memanipulasi atau memelihara suatu data**. Dibawah ini adalah cara untuk menghubungkan Java dengan database microsoft access, oracle dan mysql.

Java Database Connectivity adalah versi ODBC yang dibuat oleh Sun Microsystem. Cara kerjanya mirip tapi JDBC sepenuhnya dibangun dengan java API sehingga ia dapat dipakai cross platform sedangkan ODBC dibangun dengan bahasa C sehingga ia hanya dapat dibangun pada platform spesifik. Seperti ODBC, JDBC didasarkan pada **X/Open SQL Level Interface (CLI)**.

Dengan JDBC API, Kita dapat mengakses database dari vendor-vendor ternama seperti Oracle, Sybase, Informix dan Interbase. Untuk itu, diperlukan *driver* yang akan dipakai untuk mengakses *database* di server dari dalam program Kita (*client*). Setiap server dari vendor berbeda memiliki *driver* yang berbeda. *Driver* ini dapat Kita *download* dari situs <http://java.sun.com/products/jdbc> atau *download* dari situs vendor yang bersangkutan.

Dengan JDBC, Kita dapat melakukan mengirimkan perintah-perintah SQL ke RDBMS. Kelas-kelas serta *interface* JDBC API dapat diakses dalam paket Java, `sql(core API)` atau `javax.sql(Standard Extension API)`.

Berikut adalah Contoh Sederhana Implementasi JDBC:

```
import java.sql.*;

public class JDBC {
    static final String DB_URL = "jdbc:mysql://localhost/db_penjualan";
    static final String DB_USER = "root";
    static final String DB_PASS = "";
    static Connection conn;
    static Statement stmt;
    static ResultSet rs;

    public static void main(String[] args) {
        // INSERT
        try {
            conn = DriverManager.getConnection(DB_URL, DB_USER, DB_PASS);
            stmt = conn.createStatement();
            String kode = "AB350M GAMING 3";
            String nama = "MOTHERBOARD";
            String harga = "1700000";
            String sql = "INSERT INTO tabel_penjualan
VALUES ('"+kode+"', '"+nama+"', '"+harga+"')";
            stmt.executeUpdate(sql);
            stmt.close();
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        // SELECT
        try {
            conn = DriverManager.getConnection(DB_URL, DB_USER, DB_PASS);
            stmt = conn.createStatement();
            String sql = "SELECT * FROM tabel_penjualan";
```

```

        rs = stmt.executeQuery(sql);
        while(rs.next()){
            System.out.println("Kode: " + rs.getString("kode"));
            System.out.println("Nama: " + rs.getString("nama"));
            System.out.println("Harga: " + rs.getInt("harga"));
        }
        stmt.close();
        conn.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

Output program:

```

Kode: AB350M GAMING 3
Nama: MOTHERBOARD
Harga: 1700000

```

Contoh diatas menggunakan **MySQL** sebagai databasenya dimana username root dan password tidak diset serta berjalan di localhost dan dengan struktur tabel seperti dibawah:

```

CREATE DATABASE db_penjualan

CREATE TABLE tabel_penjualan(
    kode varchar(20),nama varchar(20),harga int
);

```

Dari contoh diatas, terdapat bagian-bagian tersendiri dalam JDBC agar pengaksesan *database* dapat berjalan dengan baik, yaitu:

1) Import Library Database

Pada Netbeans 6.5, sudah terdapat beberapa *driver* yang dapat digunakan langsung seperti Java DB, MySQL, dan PostgreSQL. Cara penggunaannya yaitu dengan **mengklik kanan** pada **folder library** dalam project yang dibuat kemudian pilih **Add Library**, lalu pilih **driver** yang akan Kita pakai, dalam contoh di atas, library yang digunakan adalah **MySQL JDBC Driver**.

2) Koneksi Ke Database dengan menggunakan URL:

```

Connection var koneksi =
DriverManager.getConnection("nama_URL","nama_user","password");

```

Tiap database mempunyai *nama_URL* yang berbeda-beda yaitu :

Pada access → jdbc:odbc:nama_database

Pada Oracle → jdbc:oracle:thin:localhost:1521:ORCL

Pada mysql → jdbc:mysql://localhost/nama_database

3) Penggunaan Database

Statement

Statement merupakan suatu kelas pada Java yang digunakan **untuk mengeksekusi suatu query**, digunakan setelah inisialisasi variabel koneksi pada *Connection* dipoint ke-2. Inisialisasinya:

Misalkan : **variabel** koneksinya yaitu **conn** , maka:

```

Statement var_statement = conn.createStatement();

```

Setelah variabel statement diinisialisasi, tentu *query* yang dilakukan harus dieksekusi agar dapat berinteraksi dengan database, untuk *query Insert, Update, dan Delete* pada umumnya menggunakan perintah:

```
stmt.executeUpdate("Query yang dilakukan");
```

Dan untuk *query Select* pada umumnya menggunakan perintah:

```
stmt.executeQuery("Query yang dilakukan");
```

Pada statement *executeQuery*, *output query* yang didapatkan adalah **suatu baris record** pada **database**, dengan kata lain merupakan **pointer** yang akan **menunjuk** suatu **baris** pada **table**.

Caranya yaitu:

```
String sql = "SELECT * FROM tabel_penjualan";  
rs = stmt.executeQuery(sql);  
while(rs.next()) {  
    System.out.println("Kode: " + rs.getString("kode"));  
    System.out.println("Nama: " + rs.getString("nama"));  
    System.out.println("Harga: " + rs.getInt("harga"));  
}
```

Untuk **mengakhiri** suatu transaksi dengan **menggunakan query** dengan cara **menutup variabel statement** yang sudah diinisialisasi.

Misal : variabel statementnya yaitu **stmt**, maka:

```
stmt.close();  
DriverManager.getConnection("nama_URL","nama_user","password");
```

4) Mengakhiri Koneksi

Untuk menjaga **keamanan data** maka perlu **mengakhiri koneksinya** dengan cara (misal variabel koneksinya **conn**):

```
conn.close();  
DriverManager.getConnection("nama_URL","nama_user","password");
```

Modul 14 Tugas Besar

Tujuan Praktikum

1. Mengimplementasikan JDBC ke dalam suatu program java dengan GUI

14.1 Implementasi JDBC dengan GUI

Pada modul 14 ini akan dicontohkan implementasi JDBC dan GUI dengan studi kasus db_penjualan seperti pada modul 13. Berikut adalah contoh sederhana dari implementasi JDBC dan GUI :

```
import java.sql.*;
import javax.swing.*;

/**
 *
 * @author IFLAB
 */
// Kelas Database
public class Database {
    static final String DB_URL = "jdbc:mysql://localhost/db_penjualan";
    static final String DB_USER = "root";
    static final String DB_PASS = "";
    static Connection conn;
    static Statement stmt;
    static ResultSet rs;

    public Database() throws SQLException{
        try{
            conn = DriverManager.getConnection(DB_URL,DB_USER,DB_PASS);
            stmt = conn.createStatement();
        }
        catch(Exception e){
            JOptionPane.showMessageDialog(null,""+e.getMessage(),"Connection
            Error",JOptionPane.WARNING_MESSAGE);
        }
    }

    public ResultSet getData(String SQLString){
        try{
            rs = stmt.executeQuery(SQLString);
        }
        catch(Exception e){
            JOptionPane.showMessageDialog(null,"Error
            :"+e.getMessage(),"Communication Error",
            JOptionPane.WARNING_MESSAGE);
        }
        return rs;
    }

    public void query (String SQLString){
        try{
            stmt.executeUpdate(SQLString);
        }
        catch(Exception e){
            JOptionPane.showMessageDialog(null,"Error
            :"+e.getMessage(),"Communication Error",
            JOptionPane.WARNING_MESSAGE);
        }
    }
}
```

```

// Kelas Penjualan
import java.sql.*;
import java.util.ArrayList;

public class Penjualan {
    private String kode;
    private String nama;
    private String harga;

    public Penjualan(){

    }

    public Penjualan(String kode, String nama, String harga) {
        this.kode = kode;
        this.nama = nama;
        this.harga = harga;
    }

    public String getKode() {
        return kode;
    }

    public String getNama() {
        return nama;
    }

    public String getHarga() {
        return harga;
    }

    public void insert_penjualan() throws SQLException{
        Database db = new Database();
        String sql = "insert into tabel_penjualan values ('"+this.getKode()+"', '"+this.getNama()+"', '"+this.getHarga()+"')";
        db.query(sql);
    }
}

//Kelas GUI
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JOptionPane;
import javax.swing.table.DefaultTableModel;
import javax.swing.text.View;

public class UI_penjualan extends javax.swing.JFrame{
    DefaultTableModel table_penjualan;

    public UI_penjualan() {
        initComponents();

        String [] judul_barang = {"Kode", "Nama", "Harga"};
        table_penjualan = new DefaultTableModel(judul_barang,0);
        table_barang.setModel(table_penjualan);
        ResetDaftar_Barang();
        show_tablebarang();
    }

    private void tambah_barangActionPerformed(java.awt.event.ActionEvent evt) {
        try {
            String kode = kode_barang.getText();
            String nama = nama_barang.getText();

```

```

        String harga = harga_barang.getText();
        Penjualan p = new Penjualan(kode,nama,harga);
        p.insert_penjualan();
        javax.swing.JOptionPane.showMessageDialog(null, "Data Berhasil
        Diinputkan");
        ResetDaftar_Barang();
        show_tablebarang();
    } catch (SQLException ex) {
        Logger.getLogger(UI_penjualan.class.getName()).log(Level.SEVERE,
        null, ex);
    }
}

private void show_tablebarang(){
    int row = table_barang.getRowCount();
    for(int i = 0; i<row; i++){
        table_penjualan.removeRow(0);
    }
    try{
        Database db = new Database();
        String sql = "select * from tabel_penjualan";
        ResultSet rs = db.getData(sql);
        while(rs.next()){
            String data[] = {rs.getString("kode"), rs.getString("nama"),
            rs.getString("harga")};
            table_penjualan.addRow(data);
        }
    }
    catch(SQLException ex){
        Logger.getLogger(View.class.getName()).log(Level.SEVERE, null, ex);
    }
}
}

```

Output dari program diatas adalah sebagai berikut :

Kode	Nama	Harga
AB350M GAMING 3	MOTHERBOARD	1700000

Gambar 14-1 Contoh implementasi sederhana dari JDBC dan GUI

Daftar Pustaka

- [1] Anonim. 2012. *Modul Praktikum Pemrograman Berorientasi Objek*. Fakultas Informatika. Institut Teknologi Telkom, Bandung.
- [2] Barclay, K. & J.Savage.2004.*Object-Oriented Design with UML and Java*. Massachusetts:Elsevier Butterworth-Heinemann.
- [3] Bruegge , Bernd & Allen H. Dutoit .2010. *Object-Oriented Software Engineering Using UML, Patterns, and Java™ Third Edition*.Pennsylvania: Prentice Hall.
- [4] Deitel, H.M & Deitel,P.J. 2004. *Java How to Program Sixth Edition*. New Jersey: Prentice Hall.
- [5] Sierra, Kathy & Bates, Bert. 2005. *Head First Java, 2nd Edition*. Sebastopol: O'Reilly Media.
- [6] Sierra, Kathy & Bates, Bert. 2008. *SCJP Sun Certified Programmer for Java™ 6 Study Guide*. New York: McGraw-Hill
- [7] Edward Hill, Jr. 2005. *Learning to Program Java*. New York: iUniverse.
- [8] Java Tutorial. <https://beginnersbook.com/java-tutorial-for-beginners-with-examples/>. 28 Desember 2018.
- [9] Java Tutorial. <https://tutorialspoint.com/java/>. 28 Desember 2018.
- [10] Java Tutorial. <https://w3schools.com/java>. 28 Desember 2018.
- [11] Java SE Documentations. <http://docs.oracle.com/javase>. 28 Desember 2018.



Kontak Kami :



@fiflab



Praktikum IF LAB



informaticslab@telkomuniversity.ac.id



informatics.labs.telkomuniversity.ac.id