1 PEMROGRAMAN DAN SIKLUS HIDUP PERANGKAT LUNAK

I.1 Pendahuluan

Perangkat lunak adalah satu bagian tak terpisahkan dari sistem komputer saat ini. Perkembangan teknologi komputer saat ini dapat dipastikan mengikutsertakan perkembangan perangkat lunak.

Perkembangan pembangunan perangkat lunak mengalami kemajuan yang signifikan sejak 6 dekade terakhir : dimulai dari sekedar memberi instruksi biner ke sistem komputer sederhana, pembuatan bahasa pemrograman tingkat rendah, tingkat menengah sampai tingkat tinggi. Selain itu, karena semakin kompleksnya kebutuhan akan komputer sebagai alat bantu komputasi, perkembangan perangkat lunak juga telah melahirkan konsep-konsep pemrograman, mulai dari konsep pemrograman sederhana (hanya menuliskan baris-baris program dari yang berjalan dari awal sampai akhir program), pembuatan prosedur-prosedur, sampai pada pemrograman berorientasi obyek.

Pemrograman berorientasi obyek merupakan konsep pemrograman yang relatif baru, di mana pemrograman diarahkan ke paradigma pembentukan obyek-obyek yang saling berinteraksi. Selain konsepnya lebih mudah dicerna oleh pemrogram, baik yang awam sekalipun, konsep pemrograman berorientasi obyek mempermudah *maintenance* software sehingga software menjadi lebih fleksibel apabila akan direvisi atau dikembangkan.

I.2 Pemrograman

I.2.1 Pengertian

Pemrograman merupakan proses "penanaman" instruksi ke dalam komputer sedemikian rupa sehingga dalam operasinya, komputer tersebut akan mengacu kepada instruksi yang telah diberikan.

Proses pemrograman akan menghasilkan suatu produk yang disebut "program". Program yang berbeda akan menyebabkan komputer memberikan hasil yang berbeda untuk suatu input yang sama. Contohnya, anggap saja kita mempunyai 2 buah program Java sebagai berikut :

a. Penjumlahan.class, yaitu suatu program yang akan menjumlahkan 2 buah input yang diberikan kepadanya, dan mencetaknya ke layar monitor.

b. Perkalian.class, yaitu suatu program yang akan mengalikan 2 buah input yang diberikan kepadanya, dan mencetaknya ke layar monitor.

Misalnya kita memasukkan pasangan 2 buah input { 1, 5 } ke dalam masing-masing program. Maka :

a. Penjumlahan.class akan mencetak nilai "6" ke layar monitor
b. Perkalian.class akan mencetak nilai "5" ke layar monitor

1.2.2 Bahasa Pemrograman

Instruksi yang diberikan kepada komputer pada dasarnya adalah instruksi berupa binary code, yaitu rangkaian kode-kode biner (yang terdiri atas bilangan-bilangan "0" dan "1") yang dapat "dimengerti" oleh komputer. Instruksi biner yang dapat "dimengerti" oleh komputer adalah instruksi yang memiliki relasi dengan operasi elementer yang dapat dilakukan oleh komputer, misalnya operasi "menyimpan 1 byte ke alamat tertentu pada memory", "memutar hard-disk", dan sebagainya. Setiap operasi pada komputer memiliki kode biner tertentu.

Pada perkembangan awal sistem komputer, para pemrogram (programmer) komputer melakukan pemrograman dengan cara memasukkan nilai-nilai biner ke dalam memory komputer. Nilai-nilai biner yang dimasukkan menggambarkan algoritma dari operasi yang harus dilakukan oleh komputer. Cara ini memiliki beberapa kesulitan, antara lain :

a. Sebelum memasukkan kode biner yang benar, programmer harus melakukan pengecekan terhadap pemetaan kode biner tersebut dengan operasi komputer yang diinginkan.
b. Ketika terjadi error, programmer harus melakukan kerja ekstra berupa pengecekan kode yang salah dan pemetaan-ulang terhadap hubungan antara kode biner dengan operasi komputer.

Seiring dengan semakin berkembangnya arsitektur komputer dan semakin banyaknya operasi yang dapat dilakukan komputer, metode penanaman instruksi dengan menggunakan binary code mulai dirasakan tidak praktis, karena sulit diterjemahkan ke dalam bahasa manusia. Oleh karena itu dibuatlah suatu perangkat yang dapat mengkonversi instruksi dari instruksi yang dimengerti oleh bahasa manusia menjadi instruksi yang dimengerti oleh mesin / komputer. Sebagai contoh :

a. Instruksi yang dimengerti oleh manusia : "b = 1 + 4;"
b. Instruksi yang dimengerti oleh komputer: "...1001101010101011100..."

Dari sini mulai muncul istilah "software" yang pada intinya merupakan satu perangkat yang berperan sebagai pengatur eksekusi operasi-operasi pada komputer.

Software-software yang dibuat sampai saat ini ditulis dengan menggunakan "bahasa pemrograman". Bahasa pemrograman dapat didefinisikan sebagai satu kumpulan pola-pola instruksi yang dapat dimengerti oleh manusia, yang digunakan untuk menulis program / aplikasi / software untuk menghasilkan operasi tertentu pada komputer.

1.2.3 Level Bahasa Pemrograman

Pengertian "bahasa yang dapat dimengerti oleh manusia" sebenarnya merupakan pengertian relatif. Bagi programmer-programmer binary code, baris code "..1001100101011..." mungkin dapat dimengerti. Tetapi bagi programmer-programmer yang terbiasa dengan *mnemonic* , baris code seperti "MOV A,#25" masih dapat dimengerti. Dan bagi programmer-programmer Java, baris code seperti "String s = new String("Hello World");" lebih dapat dimengerti daripada binary atau *mnemonic*.

Di sini muncul pemeringkatan (*levelling*) bahasa pemrograman. Bahasa pemrograman yang lebih rendah merupakan bahasa yang "cenderung lebih dimengerti oleh komputer". Sedangkan bahasa yang lebih tinggi merupakan bahasa yang "cenderung lebih dimengerti oleh manusia". Terdapat beberapa tingkat bahasa pemrograman, antara lain :

a. Bahasa tingkat rendah (*low-level language)*, misalnya bahasa mesin (binary), dan *assembler*
b. Bahasa tingkat menengah (*medium-level language*), misalnya bahasa C / C++, Fortran.
c. Bahasa tingkat tinggi (*high-level language*), misalnya bahasa Pascal.
d. Bahasa tingkat lebih tinggi (*higher-level language*), misalnya bahasa Java, DotNet.

Ciri khas dari bahasa yang levelnya lebih rendah adalah kemampuan user untuk memanipulasi operasi pada level hardware (misalnya, mengisi, mengedit, dan menghapus data pada memory dan register) lebih tinggi. Sedangkan pada bahasa yang levelnya lebih tinggi, kemampuan user untuk memanipulasi operasi pada level hardware lebih rendah. Bahkan pada bahasa *higher-level* seperti Java, user benar-benar tidak dapat melakukan manipulasi pada level hardware secara langsung, karena operasi-operasi pada hardware (misalnya pengalokasian memory, penghapusan data pada memory) sudah dilakukan secara otomatis oleh *Java Virtual Machine (JVM)*.

I.3 Siklus Hidup Perangkat Lunak

Pembuatan perangkat lunak / *software* dilakukan melalui tahapan-tahapan yang membentuk suatu siklus hidup / *lifecycle*. Siklus hidup ini dimulai dari 6 tahap, yaitu :

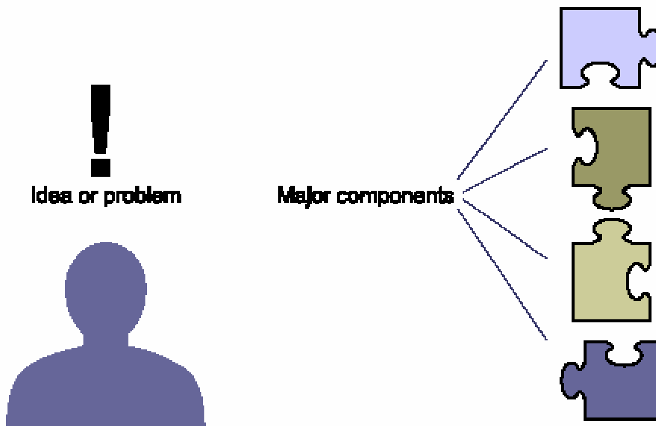
a. Analisis
b. Desain
c. Pengembangan
d. Pengujian
e. Pemeliharaan
f. Akhir Siklus

Penjelasan tentang keenam langkah tersebut dijelaskan pada bagian lain dari bab ini.

I.3.1 Analisis (Analysis)

Langkah pertama yang dilakukan dalam Analisis adalah proses investigasi terhadap masalah yang akan dipecahkan melalui produk / program yang akan dihasilkan. Pada tahap Analisis terdapat 2 hal yang harus dilakukan, antara lain :

a. Mendefinisikan permasalahan yang akan dipecahkan, pangsa pasar yang akan diambil, atau sistem yang akan dibuat. Proses ini disebut *scoping*.
b. Identifikasi komponen / sub-komponen inti yang akan menjadi komponen dasar bagi produk yang akan dihasilkan.



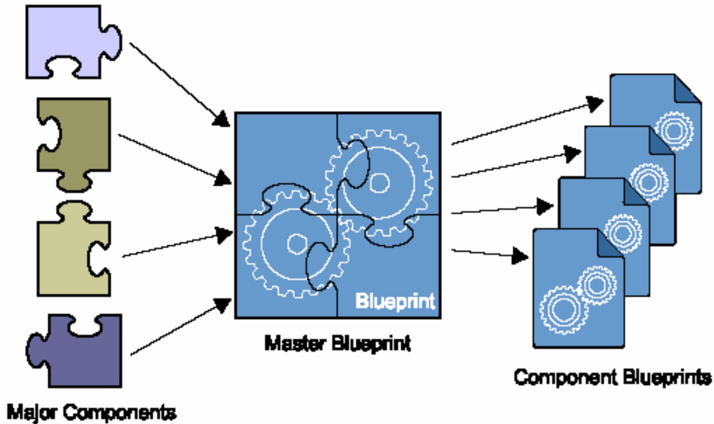
Gambar 1.1 Komponen-komponen Penyusun Produk untuk Penyelesaian Suatu Masalah

Gambar 1.1 memperlihatkan ilustrasi bahwa ketika kita mempunyai suatu permasalahan / ide, kita harus juga melakukan identifikasi terhadap komponen-komponen dasar dari produk yang akan dihasilkan.

I.3.2 Desain (Design)

Desain adalah proses implementasi hal-hal yang telah dianalisis dalam fase Analisis, misalnya komponen-komponen penyusun produk ke dalam suatu cetak biru / *blueprint*. Fase ini menghasilkan :

a. Cetak biru sistem yang akan dihasilkan
b. Spesifikasi teknis dari sistem yang akan dihasilkan



Gambar 1.2. Desain Cetak-Biru

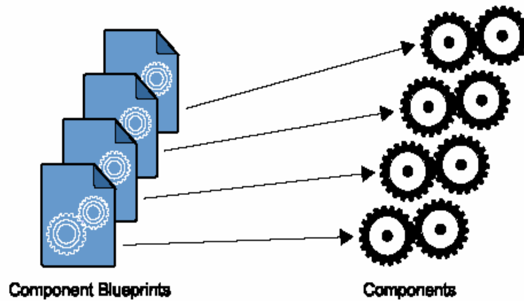
Pada Gambar 1.2, komponen-komponen inti (*Major Components*) yang telah diidentifikasi pada fase Analisis dimasukkan ke dalam desain menjadi sebuah cetak biru utama (*Master Blueprint*). Pada cetak biru utama, didefinisikan pula spesifikasi dari setiap komponen tersebut, sehingga menjadi cetak-biru beberapa komponen (*Component Blueprints*) yang memiliki pola spesifikasi yang lebih jelas.

***Catatan* : Cetak biru komponen (Component Blueprints) masih berupa *konsep*.**

I.3.3 Pengembangan (Development)

Fase pengembangan merupakan rangkaian aktivitas penggunaan cetak-biru komponen untuk membuat komponen-komponen aktual. Dalam prakteknya, dapat saja cetak-biru komponen tersebut direalisasikan dengan membuat sub-sub komponen.

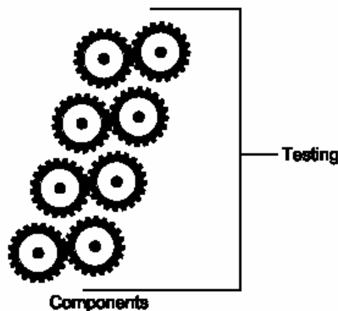
Ilustrasi mengenai fase pengembangan dapat dilihat pada Gambar 1.3.



Gambar 1.3. Pengembangan Komponen

I.3.4 Pengujian (Testing)

Pengujian adalah proses evaluasi atas komponen-komponen sampai kepada sistem secara keseluruhan apakah memenuhi kebutuhan yang diinginkan (sesuai dengan spesifikasi yang diharapkan).



Gambar 1.4 Proses Pengujian Komponen

Ada beberapa jenis pengujian yang dapat dilakukan, antara lain :

a. Unit Testing, yaitu pengujian atas semua komponen / sub-komponen penyusun produk. Hal-hal yang diuji misalnya kesesuaian output yang dihasilkan dengan output yang diharapkan, kemampuan komponen untuk meminimalisasi *bug* (gangguan), kemampuan komponen untuk mengantisipasi input yang tidak sesuai (misalnya jika input diharuskan berupa bilangan, ternyata yang dimasukkan oleh *user* adalah huruf).

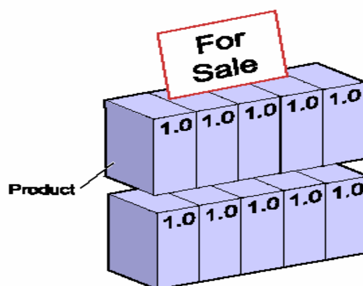
b. Functional Testing, yaitu pengujian atas kesesuaian fungsi yang dihasilkan oleh komponen-komponen dengan fungsi yang diinginkan. Misalnya jika sebuah komponen diharapkan dapat melakukan perhitungan "pendapatan bersih" yang merupakan hasil pemotongan-pemotongan terhadap "pendapatan kotor", maka functional testing atas komponen tersebut akan sukses jika komponen tersebut dapat menghasilkan hasil perhitungan yang benar.
c. Flow Graph Testing, yaitu pengujian alur proses pada prototype / produk yang dihasilkan. Testing ini bertujuan menguji kesesuaian antara *event* dengan perpindahan *state*.
d. Performance Testing, yaitu pengujian kinerja proses-proses yang dijalankan oleh prototype / produk. Kinerja diukur berdasarkan parameter-parameter yang ditetapkan sebelumnya, misalnya kecepatan proses, efisiensi penggunaan memory, dan lain-lain.
e. Security Testing, yaitu pengujian keamanan sistem. Biasanya pengujian ini dilakukan jika spesifikasi produk mensyaratkan adanya pembatasan hak akses atas sekumpulan data.
f. Integration Testing, yaitu pengujian interaksi antar komponen pada produk.

I.3.5 Implementasi (Implementation)

Implementasi adalah proses membuat produk dapat digunakan oleh *user*. Proses ini terdiri atas beberapa langkah, antara lain :

a. Instalasi produk ke sistem / komputer *user*
b. Pelatihan kepada *user* dengan tujuan *user* dapat menggunakan produk tersebut.

Catatan : Fase implementasi kadang-kadang dilakukan paralel dengan pengujian, terutama untuk memastikan kompatibilitas / kesesuaian produk jika diinstalasi di sistem milik user.

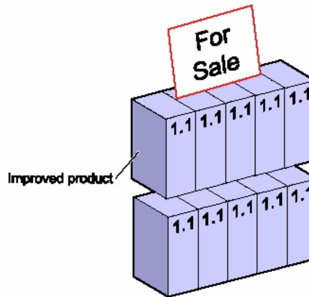


Gambar 1.5. Implementasi Produk

I.3.6 Pemeliharaan (Maintenance)

Pemeliharaan terdiri atas proses-proses perbaikan terhadap masalah-masalah yang terjadi pada produk. Pemeliharaan terhadap produk dapat berupa :

a. Perbaikan bug
b. Re-install produk ke sistem komputer
c. Back-up data
d. Perbaikan pada komponen-komponen
e. dan lain-lain.



Gambar 1.6. Pemeliharaan

I.3.7 Akhir Siklus (End-Of-Life)

Fase EOL ini mencakup proses-proses yang mengumpulkan kesimpulan dan permintaan *user* terhadap produk yang telah berjalan di sistem. Hasil *feedback* ini dapat menghasilkan keputusan untuk memodifikasi produk sehingga dihasilkan versi-versi produk yang lebih mutakhir dan lebih dapat memenuhi kebutuhan *user*.

Gambar 1.7. End-Of-Life

2 ANALISIS MASALAH BERORIENTASI OBYEK

II.1 Pendahuluan

Pada Modul II ini, akan dibahas bagaimana menganalisis suatu masalah dengan menggunakan konsep analisis berbasis obyek (Object-Oriented Analysis). Analisis masalah akan menghasilkan cetak-biru dari komponen-komponen pembentuk obyek.

II.2 Contoh Masalah

Contoh masalah yang akan dibuat desain pemecahannya adalah sebagai berikut :

> Sebuah koperasi menginginkan sebuah aplikasi pelaporan rekapitulasi stok barang yang mencatat persediaan, pemasukan dan pengeluaran barang. Laporan ini dilakukan seminggu sekali.
>
> Aplikasi ini akan dijalankan oleh seorang petugas khusus logistik. Petugas tersebut akan mencatat Stok barang didapat dari supplier-supplier, baik dari dalam maupun luar kota. Penjualan barang dilakukan pada sebuah mini market, yang pembelinya berasal dari anggota koperasi maupun yang bukan anggota koperasi.
>
> Kemudian ada kebijakan kalau barang berupa makanan yang masa kedaluwarsanya sudah lewat, harus dibuang, dan yang masa kedaluwarsanya tinggal 6 bulan lagi, diberikan diskon 50% (asumsi makanannya semuanya mempunyai masa kedaluwarsa > 1 tahun). Kebijakan lainnya adalah : untuk item barang yang tinggal <= 40% persen dari seharusnya, harus segera ditambah stoknya.

STOK BARANG KOPERASI

Minggu ke :
Bulan ke :

ID barang	Stok	Kedaluwarsa	Potongan	Harga

DATA BARANG

ID Barang	Supplier	Harga Dasar

II.2 Identifikasi Domain Masalah

Domain masalah didefinikan sebagai ruang lingkup permasalahan yang akan dipecahkan. Ruang lingkup ini membatasi permasalahan sehingga permasalahan tidak meluas, dan tetap pada fokusnya.

Domain masalah ini dapat ditentukan dari permintaan / *requirements* yang diberikan oleh pihak yang meminta aplikasi (client). Setelah permintaan tersebut dikumpulkan, maka developer (dalam hal ini programmer) dapat membuat pernyataan tentang apa yang akan dibuatnya , misalnya : "Membuat sistem pelaporan stok barang pada koperasi yang mengakomodasi kebijakan-kebijakan kedaluwarsa dan potongan harga" .

II.3 Identifikasi Obyek

Setelah domain masalah diidentifikasi, developer sudah dapat melakukan identifikasi terhadap obyek-obyek yang akan berinteraksi untuk memecahkan masalah.

Langkah pertama untuk mengidentifikasi obyek adalah melakukan identifikasi atas beberapa sifat dari obyek, antara lain :

a. Obyek dapat bersifat konseptual atau berupa benda fisik. Untuk masalah "Sistem Manajemen Stok Barang", obyek yang bersifat konseptual misalnya *Stok Barang*. Sedangkan obyek yang berupa benda fisik misalnya *Barang*.
b. Obyek memiliki atribut atau karakteristik. Karakteristik ini berupa data yang dibawa oleh obyek tersebut. Untuk obyek *Stok Barang*, karakteristiknya antara lain *jumlah barang, nama barang*.
c. Obyek memiliki operasi. Operasi ini menggambarkan aktivitas-aktivitas yang dapat dilakukan oleh obyek ini. Biasanya operasi memberikan efek terhadap karakteristik, yaitu berupa modifikasi nilai. Misalnya untuk obyek *Barang*, operasinya adalah : *tambahBarang*.

Dalam fase ini, kegiatan utamanya adalah mengumpulkan obyek-obyek yang mungkin terlibat sebanyak-banyaknya. Kemudian daftarkan atribut dan operasi yang dimiliki oleh setiap obyek.

Sebagai contoh, obyek-obyek yang akan diperkirakan akan digunakan dalam memecahkan masalah dapat dilihat pada Tabel II.1.

Tabel II.1
Obyek yang Diperkirakan Akan Digunakan Dalam Aplikasi

Obyek	Atribut	Operasi
Barang	a. jumlah	a. hitung tanggal kedaluwarsa
	b. harga beli	b. hitung harga jual
	c. tanggal kedaluwarsa	c. hitung diskon
	d. harga jual	
	e. ID barang	
	f. diskon	

Obyek	Atribut	Operasi
Supplier	a. nama b. lokasi c. ID supplier d. jenis barang	a. modifikasi nama supplier b. modifikasi lokasi supplier
Koperasi	a. daftar transaksi b. gudang	a. modifikasi daftar transaksi
Pembeli	a. ID pembeli b. nama pembeli	a. modifikasi ID pembeli b. modifikasi nama pembeli
Gudang	a. daftar stok barang b. koperasi	a. modifikasi stok barang b. set koperasi
Daftar Transaksi	a. data penjualan [ ] b. data pembelian [ ]	a. tambah data penjualan b. tambah data pembelian
Daftar Stok Barang	a. Barang [ ] b. jumlahBarang[]	a. tambah barang b. kurangi barang

II.4 Seleksi Obyek

Langkah berikutnya dalam menganalisis masalah adalah melakukan seleksi terhadap obyek-obyek. Seleksi obyek dilakukan dengan mempertimbangkan hal-hal berikut :

a. Relevansi dengan permasalahan

Untuk menentukan apakah sebuah obyek relevan dengan permasalahan, haruslah dievaluasi melalui beberapa pertanyaan berikut :

a) Apakah obyek tersebut eksis pada batasan-batasan yang ditentukan pada permasalahan ?
b) Apakah obyek tersebut dibutuhkan untuk menyelesaikan masalah ?
c) Apakah sebuah user dibutuhkan sebagai bagian dari interaksi antara *user* dan solusi ?

b. Eksistensi obyek haruslah independen

Masalah independensi obyek ini terjadi jika ada obyek yang menjadi atribut obyek lainnya. Harus dievaluasi apakah kehadiran salah satu obyek mewajibkan kehadiran obyek lain pula ? Hal ini terjadi pada saat Obyek A memiliki atribut berupa Obyek B, dan Obyek B memiliki atribut berupa Obyek A.

Solusi dari permasalahan ini adalah memilih salah satu obyek untuk menghilangkan salah satu atribut yang menyebabkan hilangnya independensi, dan melakukan re-design obyek yang mana dari kedua obyek tersebut yang benar-benar dibutuhkan, dan menetapkan atribut-atribut yang tepat dari obyek yang dipilih.

Sekarang akan dicoba memilih obyek-obyek dari daftar obyek yang telah dibuat :

a. *Barang*

Obyek *Barang* mendeskripsikan informasi tentang barang yang menjadi item yang dijual di koperasi, seperti *jumlah, harga beli, tanggal kedaluwarsa,* dll. Karena informasi yang dikandungnya relevan dengan permasalahan, yang membutuhkan informasi jumlah barang, tanggal kedaluwarsa, dan harga, maka *Barang* dapat menjadi obyek utama untuk menyelesaikan masalah.

b. *Supplier*

Obyek *Supplier* mendeskripsikan informasi tentang supplier barang yang menjadi acuan koperasi untuk membeli barang, misalnya *ID supplier, jenis barang*, dll. Menurut spesifikasi desain, supplier tidak termasuk ke dalam *scope* masalah, karena masalah lebih banyak berkutat di bagian penentuan harga barang. Oleh karena itu, obyek *Supplier* dapat tidak diikutsertakan.

c. *Koperasi*

Obyek *Koperasi* mendeskripsikan informasi berupa daftar transaksi dan gudang. Kedua informasi tersebut

merupakan obyek-obyek yang memuat informasi tersendiri. Di sini terdapat obyek yang menjadi atribut obyek lain (*Daftar Transaksi* dan *Gudang* menjadi atribut *Koperasi*).

Jika dianalisa relevansinya dengan permasalahan, class *Koperasi* dibutuhkan sebagai class utama (main class), karena permasalahan yang akan dipecahkan berada pada lingkungan koperasi.

Jika dianalisa tingkat independensinya terhadap class lain, terutama yang menjadi atributnya, maka class *koperasi* memiliki independensi terhadap class *daftar transaksi*. Tetapi class *koperasi* memiliki ketergantungan dengan class *gudang*. Hal ini disebabkan oleh masuknya *gudang* sebagai atribut dari *koperasi* dan masuknya *koperasi* sebagai atribut dari *gudang*.

Karena antara *Koperasi* dan *Gudang* terdapat ketergantungan, solusinya adalah menghilangkan *Gudang* dan memasukkan data yang dimilikinya ke *Koperasi,* yaitu *daftar stok barang*.

d. *Pembeli*

Pembeli mendeskripsikan informasi berupa data pembeli yang akan membeli barang dari koperasi. Data pembeli tersebut adalah *ID pembeli* dan *nama pembeli*.

Jika dianalisa relevansinya terhadap permasalahan, class *Pembeli* tidak perlu digunakan, karena permasalahan tidak menyentuh perihal pencatatan pembeli.

e. *Daftar Transaksi*

Daftar transaksi mendeskripsikan informasi berupa data penjualan dan data pembelian. Data penjualan dan pembelian diperlukan sebagai variabel yang akan mempengaruhi stok barang.

Jika dilihat relevansinya terhadap permasalahan, daftar transaksi ini mempunyai efek langsung terhadap stok barang. Kemudian pada daftar transaksi tidak terdapat obyek yang merupakan salah satu obyek yang telah didaftarkan.

Jadi, obyek *Daftar Transaksi* dapat diajukan menjadi obyek yang terlibat dalam aplikasi.

f. *Daftar Stok Barang*

Daftar Stok Barang mendeskripsikan informasi berupa barang-barang yang tersedia di stok, dan jumlah masing-masing barang. Karena *Daftar Stok Barang* diperlukan oleh obyek *Koperasi*, yang berarti ada relevansinya dengan permasalahan, maka dapat menjadi obyek.

II.5 Solusi

Setelah obyek-obyek diseleksi, maka didapatlah solusi obyek apa saja yang digunakan untuk menyelesaikan masalah, yaitu obyek *Barang, Koperasi, Daftar Transaksi* dan *Daftar Stok Barang*, seperti diperlihatkan pada Gambar 2.1. Perhatikan bahwa pada setiap obyek, terdapat kotak yang menunjukkan nama obyek (paling atas), nama-nama variabel, dan nama-nama operasi (paling bawah).

BARANG	KOPERASI
jumlah harga beli tanggal kedaluwarsa harga jual ID barang	daftar stok barang daftar transaksi
hitung tanggal kedaluwarsa hitung harga jual hitung diskon	modifikasi daftar transaksi
DAFTAR TRANSAKSI	**DAFTAR STOK BARANG**
data penjualan [ ] data pembelian [ ]	barang [] jumlah barang []
tambah data penjualan tambah data pembelian	tambah barang kurangi barang

Gambar 2.1. Solusi Permasalahan

3

MEMBUAT DAN MENGUJI PROGRAM JAVA

III.1. Identifikasi Komponen-komponen dari Sebuah Class

Class adalah cetak-biru dari obyek. Seperti halnya semua cetak-biru dalam kehidupan sehari-hari, class merupakan acuan bagaimana obyek itu dibentuk, data apa saja yang disimpan oleh obyek, dan operasi-operasi apa saja yang dapat dilakukan oleh obyek.

Karena pentingnya peran sebuah class dalam membangun sebuah aplikasi menggunakan konsep Pemrograman Berbasis Obyek, maka perlu dipahami struktur dasar dari class.

III.1.1 Strukturisasi Class

Struktur dasar sebuah class pada intinya ada 4 bagian, yaitu :

a. deklarasi class
b. deklarasi dan inisialisasi atribut
c. pendefinisian method (optional)
d. komentar (opsional)

III.1.2 Deklarasi Class

Untuk membuat sebuah obyek, terlebih dahulu harus dilakukan pendeklarasian class. Pendeklarasian class dimaksudkan untuk mendefinisikan data-data yang dibawa oleh obyek tersebut, lalu operasi-operasi yang dapat dilakukan.

Pada pemrograman Java, pendeklarasian class dilakukan dengan menggunakan syntax sebagai berikut :

```
[modifier] class class_identifier
```

di mana :

a. *[modifiers]* merepresentasikan keywords pada teknologi Java yang memodifikasi cara-cara penggunaan class. Contoh : *public, protected, private, static, final*.
b. *class* adalah kata kunci pada teknologi Java, yang mengindikasikan deklarasi sebuah class.
c. *class_identifier* adalah nama class yang dideklarasikan

Contoh deklarasi class adalah deklarasi class *Barang* (Contoh III.1).

Contoh III.1

```
public class Barang {

}
```

III.1.3 Deklarasi Variabel dan Penugasan

Deklarasi variabel dilakukan di dalam class. Variabel yang didefinisikan di class merupakan atribut dari class tersebut (yang otomatis merupakan atribut dari obyek). Secara rinci pendeklarasian variabel akan diuraikan pada Modul IV.

Penugasan terhadap variabel merupakan pemberian nilai kepada nilai. Karena dalam deklarasi class, penugasan variabel merupakan penugasan pertama kali, maka penugasan ini dapat juga disebut *inisialisasi variabel*.

Adapun syntax umum deklarasi dan inisialisasi adalah sebagai berikut :

```
[modifiers] data_type identifier [ = value ] ;
```

di mana :

a. [*modifiers*] merepresentasikan beberapa kata kunci pada teknologi Java, misalnya *public, private, static,* dll.
b. *data_type* merepresentasikan tipe data dari variabel, misalnya *int, String, float,* < nama class >, dll.
c. *identifiers* merepresentasikan nama variabel
d. *value* adalah nilai awal (inisialisasi) dari variabel. Pencantuman *value* ini sifatnya opsional, karena inisialisasi variabel tidak harus dilakukan bersamaan dengan deklarasi.

Contoh pendeklarasian variabel adalah pendeklarasian variabel-variabel atribut pada class *Barang* (Contoh III.2).

Contoh III.2

```
import java.util.Date;

public class Barang {
               public int jumlah ; //contoh deklarasi
               public int hargaBeli ;
               public Date tanggalKedaluwarsa ;
               public int hargaJual ;
               public String idBarang ;
               double diskon = 0.0 ; //contoh penugasan
}
```

III.1.4 Pendefinisian Method

Method merepresentasikan operasi-operasi yang dapat dilakukan oleh obyek. Dalam penulisan, yang membedakan method dan variabel adalah : method selalu diakhiri dengan ( ) atau (< nama argumen >). Pendeklarasiannya juga berbeda, yaitu dengan menambahkan blok { … } dan mengisi blok tersebut dengan baris-baris program.

Bentuk umum penulisan deklarasi method adalah sebagai berikut :

```
[modifiers] return_type method_identifier ([arguments]){
        method_code_block;
}
```

di mana :

- *a.* *[modifiers]* merepresentasikan keywords pada teknologi Java yang memodifikasi cara-cara penggunaan method. Contoh : *public, protected, private, static, final.*
- *b.* *return_type* adalah tipe nilai yang akan dikembalikan oleh method yang akan digunakan pada bagian lain dari program. *Return_type* pada method sama dengan tipe data pada variabel. *Return_type* dapat merupakan tipe data primitif maupun tipe data referensi.
- *c.* *method_identifier* adalah nama method.
- *d.* *([arguments])*, merepresentasikan sebuah daftar variabel yang nilainya dilewatkan / dimasukkan ke method untuk digunakan oleh method. Bagian ini dapat tidak diisi, dan dapat pula diisi dengan banyak variabel.
- *e.* *method_code_block*, adalah rangkaian pernyataan / *statements* yang dibawa oleh method.

Contoh pendefinisian method pada class *Barang* ditunjukkan pada Contoh III.3.

Contoh III.3

```
public class Barang {
        ... //inisialisasi variabel

        public void setIDBarang( String id ){
         idBarang = id;
        }
}
```

III.1.5 Pemberian Komentar

Pemberian komentar dilakukan untuk menandai baris-baris program tertentu dengan beberapa catatan. Catatan ini biasanya berisi maksud dari baris program tersebut, atau keterangan lainnya.

Pemberian komentar dimaksudkan agar di kemudian hari, ketika program membutuhkan perbaikan, programmer (baik programmer yang membuat program pertama kali maupun penggantinya) tidak terlalu bingung menganalisa maksud dari baris program tersebut.

Pemberian komentar dapat dilakukan dengan 2 macam syntax, yaitu :

a. Pada baris yang dikomentari ditulis : *//.......(komentar).*
   Contoh :

```
public int hargaJual ; // ini variabel untuk
                       // menyimpan data
                      //harga jual
```

b. Pada awal baris yang dikomentari ditulis : /* ..... (komentar) , lalu pada akhir baris yang dikomentari ditulis : (komentar) ..... */
   Contoh :

```
/* ini variabel untuk menyimpan
   data harga jual */
      public int hargaJual ;
```

III.2. Membuat dan Menguji Program Java

III.2.1 Konfigurasi yang Dibutuhkan

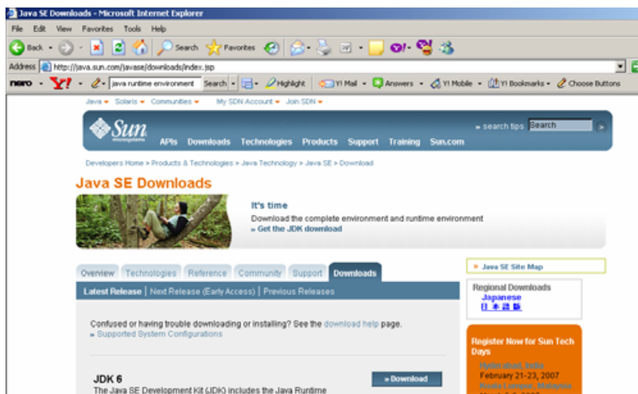
Untuk membuat dan menguji program Java, diperlukan suatu *environment* pemrograman Java, yaitu Java Runtime Environment (JRE), yaitu sebuah mesin virtual inti yang dapat menjalankan program Java dengan melakukan proses *loading* kode program, memverifikasi kode tersebut, dan mengeksekusinya.

Java Runtime Environmrnt (JRE) ini dapat di-*download* dari situs milik Sun Microsystem secara gratis, dengan prosedur sebagai berikut :

a. Masuklah ke dalam situs http://www.java.sun.com, maka akan muncul tampilan situs Sun Developer Network 9 SDN) seperti pada Gambar 3.1 (tampilan per 6 Februari 2007).

Gambar 3.1 Situs Sun Developer Network

b. Kliklah pada bagian *Popular Download : Java SE*. Maka akan terlihat tampilan *Java SE Downloads* seperti pada Gambar 3.2.



Gambar 3.2. Halaman *Java SE Downloads*

c. Kemudian klik-lah tab *Downloads*. Anda akan masuk ke dalam halaman *Sun Downloads* yang berisi file-file open source Java yang dapat didownload (Gambar 3.3).

Gambar 3.3. Halaman *Sun Downloads*

d. Pada halaman *Sun Downloads* terdapat notifikasi untuk menyetujui *agreement* untuk men-*download* file-file dari Sun Microsystems, yang berbunyi :

Required: You must accept the license agreement to download the product.
◯ **Accept** License Agreement | Review License Agreement
◯ **Decline** License Agreement

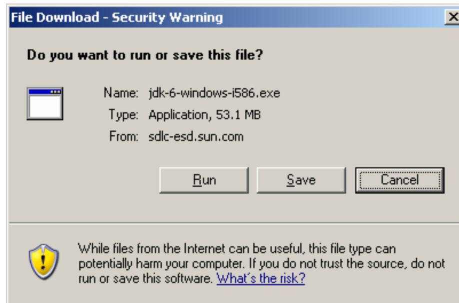
Klik *Review License Agreement*. Agreement tersebut lebih banyak membahas masalah penyebaran source. Jika merasa tidak perlu membaca License Agreement, klik saja radio button *Accept License Agreement*.

e. Halaman akan *refresh* dalam beberapa detik, dan kemudian Anda dapat mendownload Sun Development Kit (SDK) yang sudah termasuk di dalamnya Java Runtime Environment (JRE). Jika Anda menggunakan sistem operasi Windows, carilah *Windows Platform - Java (TM) SE Development Kit 6* (lihat Gambar 3.4).

Windows Platform - Java(TM) SE Development Kit 6			
☐	Windows Offline Installation, Multi-language	jdk-6-windows-i586.exe	53.16 MB
☐	Windows Online Installation, Multi-language	jdk-6-windows-i586-iftw.exe	361.62 KB

Gambar 3.4. Windows Platform - Java (TM) Development Kit 6.

f. Kliklah *Windows Offline Installation, Multi-language*. Offline installation maksudnya adalah anda akan mendownload dahulu file installer yang lengkap (jdk-6-windows-i586.exe), baru kemudian Anda menginstall dari komputer (PC / laptop) anda. Maka akan muncul message box seperti pada Gambar 3.5.

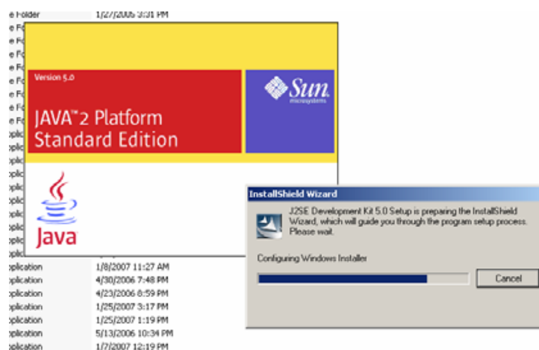


Gambar 3.5. Message Box Sebelum Download

g. Klik *Save*, lalu pilih pada folder mana akan Anda simpan file *jdk-6-windows-i586.exe* tersebut.
h. Download dimulai.

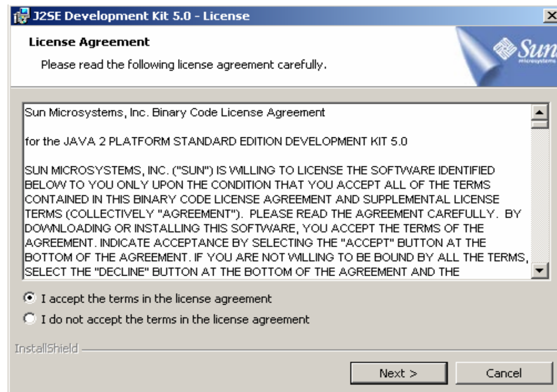
Setelah di-*download*, maka langkah berikutnya adalah melakukan instalasi. Langkah-langkahnya adalah sebagai berikut :

a. *Double-click* file *jdk-6-windows-i586.exe*. Maka akan keluar tampilan seperti pada Gambar 3.6.



Gambar 3.6. Mulai Instalasi Java 2 Standard Edition

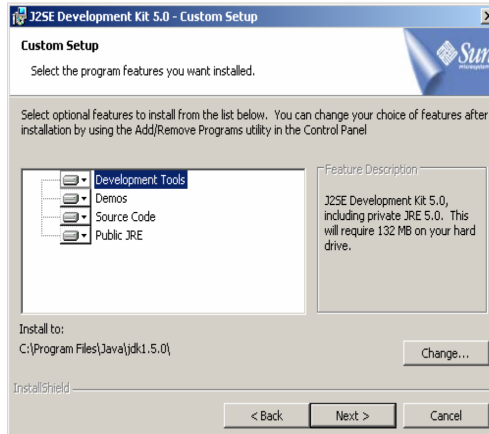
b. Kemudian akan muncul konfirmasi License Agreement, seperti pada Gambar 3.7.



Gambar 3.7. License Agreement

Klik radio button *I accept the terms in the license agreement*. Lalu klik *Next*.

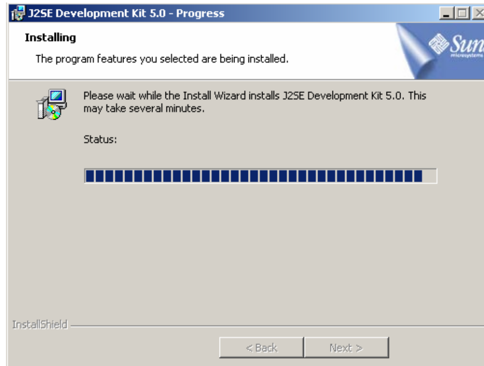
c. Kemudian akan muncul halaman *Custom Setup* seperti pada Gambar 3.8.



Gambar 3.8. Custom Setup

Klik *Change...* jika Anda ingin mengubah lokasi instalasi. Setelah Anda sudah menentukan lokasi instalasi, maka klik *Next*.

d. Setelah itu, instalasi dilakukan secara otomatis, seperti terlihat pada Gambar 3.9.



Gambar 3.9. Progress Installing

e. Hasil instalasi adalah sebuah sebuah folder j2sdk.1.6 dengan berbagai sub-folder, antara lain :
   a) /bin
   b) /lib
   c) /jre
   d) /include

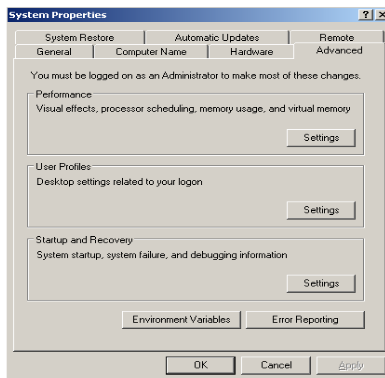
   Sementara ini yang dianggap penting adalah folder /bin. Sebab pada folder /bin terdapat file *executable* yang digunakan untuk kompilasi, yaitu *javac.exe* dan file *executable* yang digunakan untuk eksekusi, yaitu *java.exe*.

Setelah Java 2 Standard Edition diinstalasi, maka langkah berikutnya adalah melakukan *setting path*. Path diperlukan supaya ketika perintah kompilasi (*javac*) dan perintah eksekusi (*java*) dijalankan dari lokasi manapun pada *file system* , maka system akan melacak referensi perintah tersebut melalui path yang telah di-*setting* . Jika Anda menggunakan sistem operasi Windows XP, maka langkah-langkah setting path adalah sebagai berikut:

a. Pada desktop, klik *Start*, lalu klik-kanan *My Computer*, lalu klik *Properties*. Maka akan muncul frame *System Properties*, seperti pada Gambar 3.10.

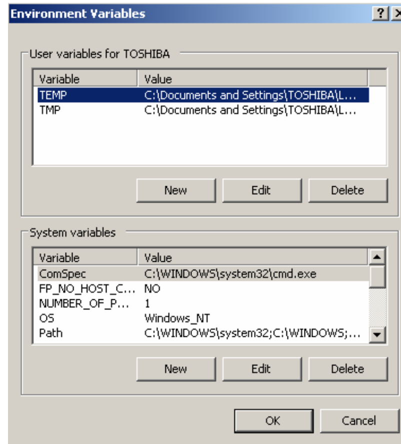
Gambar 3.10. System Properties

b. Klik tab *Advanced* pada frame *System Properties*. Akan muncul beberapa pilihan seperti pada Gambar 3.11.



Gambar 3.11. Tab Advanced pada System Properties

c. Klik *Environment Variables*. Maka akan terlihat frame *Environment Variables* yang terdiri atas variabel-variabel *user* dan variabel-variabel sistem. Variabel-variabel user adalah variabel yang bersesuaian dengan user. Variabel ini akan aktif ketika user yang login sesuai dengan peruntukan variabel tersebut. Sedangkan variabel system adalah variabel yang berlaku siapapun yang melakukan *login* ke sistem. Lihat Gambar 3.12.



Gambar 3.12. Environment Variables

d. Jika pada *System Variables* terdapat variabel *Path*, klik *Edit*, lalu tambahkan pada bagian belakang (misalnya file j2sdk berada pada c:\ j2sdk)

```
 .... ( path sebelumnya ) ; c:\j2sdk.1.6 ;
c:\j2sdk1.6\bin;
```

e. Lakukan *restart* komputer Anda. Maka setelah dilakukan *restart* maka kita dapat memanggil perintah *javac* dan *java* dari berbagai lokasi di *file system*. Dengan demikian, semua tahapan persiapan sudah selesai sampai di sini.

III.2.2 Membuat dan Menggunakan Class

Pada bagian III.5.2 ini akan dipraktekkan bagaimana cara membuat dan menggunakan class. Pada Contoh III.4, akan dibuat sebuah aplikasi sederhana yang akan menampilkan semua atribut dari class *Barang*. Aplikasi ini terdiri atas 2 class, yaitu :

a. class *Barang*, yaitu class yang menyimpan data-data barang. Class ini disimpan pada file dengan nama yang sama dengan nama class tersebut, yaitu : *Barang.java*.
b. class *DisplayBarang,* yaitu class utama yang menampilkan data-data yang disimpan oleh class *Barang*. Class ini disimpan pada file dengan nama yang sama dengan nama class, yaitu : *DisplayBarang.java*.

Program pada *Barang.java* diperlihatkan pada Contoh III.4.

Contoh III.4

```
public class Barang {
        public int jumlah ; //contoh deklarasi
        public int hargaBeli ;
        public int hargaJual ;
        public String idBarang ;
        double diskon = 0.0 ; //contoh penugasan

        public int getJumlah(){
          return jumlah;
         }
}
```

Simpanlah file *Barang.java* disimpan pada , misalnya, ***C:\Latihan\Barang.java***

Sedangkan program pada *DisplayBarang.java* adalah sebagai berikut :

```
public class DisplayBarang{
        public static void main (String[] args){
                Barang baju = new Barang();
                baju.jumlah = 25;
                baju.hargaBeli = 10000;
                baju.hargaJual = 25000;
                System.out.println(
                "Harga jual baju = " + baju.hargaJual +"\n"+
                "Harga beli baju = " + baju.hargaBeli + "\n"+
                "Jumlah baju yang akan dijual = "+
                baju.jumlah
              );
        }
}
```

Simpanlah file *DisplayBarang.java* disimpan pada , misalnya, C:\Latihan\DisplayBarang.java

III.3. Mengkompilasi dan Mengeksekusi Program

Kompilasi adalah proses konversi baris-baris program dari format text menjadi satu set bytecode. Bytecode tersebut disimpan dalam file berekstension *.class. Misalnya file *Barang.java* dikompilasi. Hasil kompilasi tersebut adalah file baru bernama *Barang.class*, yang disimpan pada folder yang sama dengan file *Barang.java* (jika proses kompilasi tidak menyebutkan lokasi file hasil kompilasi diletakkan).

Untuk mengkompilasi file *.java, maka terlebih dahulu harus masuk ke DOS prompt, melalui langkah-langkah berikut :

a. Pada desktop, klik *Start*
b. Klik *Run...*

c. Pada DOS prompt, masuk ke folder C:\Latihan> , jika Anda menyimpan file-file latihan Anda di folder tersebut. Jika tidak, masuklah ke folder latihan buatan Anda sendiri.
d. Setelah masuk ke dalam folder latihan Anda, ketikkan :

```
C:\Latihan>javac Barang.java
```

Lalu tekan *Enter*.
e. Jika setelah ditekan *Enter*, kemudian muncul prompt lagi (misalnya : ***C:\Latihan***>), maka proses kompilasi sukses.
f. Cek pada folder yang sama, apakah file *Barang.class* sudah terbentuk. Ketiklah :

```
C:\Latihan>dir
```

g. Jika terdapat file *Barang.class*, berarti proses kompilasi sudah sukses.
h. Lakukan pula kompilasi untuk file *DisplayBarang.java* dengan langkah yang sama seperti langkah d.

Setelah file-file *.java dikompilasi, maka file-file hasil kompilasi (*.class) siap dieksekusi. File yang dapat dipanggil untuk dieksekusi adalah file yang berisi method *main( )*. Dalam hal ini adalah file *DisplayBarang.class*. Prosedur eksekusi programnya adalah sebagai berikut :

a. Pada desktop, klik *Start*
b. Klik *Run...*
c. Pada DOS prompt, masuk ke folder C:\Latihan> , jika Anda menyimpan file-file latihan Anda di folder tersebut. Jika tidak, masuklah ke folder latihan buatan Anda sendiri.
d. Setelah masuk ke dalam folder latihan Anda, ketikkan :

```
C:\Latihan>java DisplayBarang
```

Lalu ketik *Enter*.

e. Output dari program adalah sebagai berikut :

```
Harga jual baju = 25000
Harga beli baju = 10000
Jumlah baju yang akan dijual = 25
```

EKSPERIMEN

1. Deklarasikan 2 buah class bebas pada satu file dengan masing-masing class memiliki variabel-variabel atribut (minimal 1 buah), dan tiap-tiap class mempunyai modifier *public*, misalnya salah satunya adalah:

```
public class Barang {
      public int id;
}
```

   Berilah nama file tersebut dengan nama salah satu class, misalnya Barang.java.
   Lakukan kompilasi. Apa yang terjadi ?

2. Kemudian salah satu class yang nama classnya bukan nama File, modifiernya dijadikan *private*, misalnya :

```
private class Koperasi{

}
```

   Simpan perubahan pada file dengan nama file yang sama dengan nama class yang bermodifier *public*.
   Lakukan kompilasi. Apa yang terjadi ?

3. Buatlah kesimpulan dari eksperimen nomor 1 dan nomor 2.

4 DEKLARASI, INISIALISASI, DAN PENGGUNAAN VARIABEL

IV.1 Identifikasi Penggunaan Variabel dan Syntax

Variabel adalah elemen penyimpanan data secara virtual. Dalam program, data dapat digunakan sebagai *operand* pada operasi-operasi aritmatika dan logika, atau sebagai parameter dalam operasi-operasi percabangan (*branching*) dan operasi berulang (*looping*).

Syntax adalah pola penulisan *statement* pada program, yang mewakili suatu instruksi tertentu. Penulisan *statement* pada setiap bahasa pemrograman mempunyai aturan syntax sendiri, tidak terkecuali untuk bahasa pemrograman Java. Dalam pembahasan selanjutnya, akan disertakan contoh-contoh *syntax*.

Dalam pembahasan lebih lanjut pada Bab IV ini, akan digunakan program pada Contoh 4.1.

Contoh 4.1

```
1   public class Mobil{
2      //deklarasi dan inisialisasi variabel anggota
3      public String noPol = "D 234 LE";
4      public String merk = "Suzuki Escudo";
5      public int harga = 70000000;
6      public int tahunPembuatan = 1999;
7      public String namaPemilik = "Sutrisno";
8      //akhir   deklarasi   dan   inisialisasi   variabel
9      anggota
10
11     //definisi method tampilkanInfoMobil( )
12     public void tampilkanInfoMobil( ){
13         //menampilkan Nomor Polisi
14         System.out.println("Nomor Polisi : " + noPol);
15         //menampilkan Merk
16         System.out.println("Merk : "+merk);
17         //menampilkan Harga
18         System.out.println("Harga : "+harga);
19         //menampilkan Tahun Pembuatan
20         System.out.println("Tahun     Pembuatan     :
21         "+tahunPembuatan);
22         //menampilkan Nama Pemilik
23         System.out.println("Nama Pemilik " +
```

```
24                                  namaPemilik);
25      }//akhir dari method tampilkanInfoMobil
26
27      //main method
28      public static void main ( String[ ] args ){
29      //instanstiasi obyek mobil1
30      Mobil mobil1 = new Mobil ( );
31      //perintah untuk menampilkan data tentang mobil1
32      mobil1.tampilkanInfoMobil( );
33      }
34   }//akhir dari class
```

Pada Contoh 4.1 diberikan contoh class *Mobil*. Class ini memiliki variabel anggota (*member variables*), yaitu :

a. **noPol**, bertipe data *String*, digunakan untuk menyimpan data nomor polisi dari mobil
b. **merk**, bertipe data *String*, digunakan untuk menyimpan data merek mobil.
c. **harga**, bertipe data *int*, digunakan untuk menyimpan data harga mobil.
d. **tahunPembuatan**, bertipe data *int*, untuk menyimpan data tahun pembuatan mobil.
e. **namaPemilik**, bertipe data *String*, untuk menyimpan data nama pemilik mobil.

Catatan : variabel anggota juga dapat disebut :

a. variabel atribut / *attribute variables*, selama tidak memiliki modifier *static*.
b. variabel instans / *instance variables*, karena variabel-variabel ini akan di-*copy* ke dalam variabel-variabel yang akan menjadi milik eksklusif dari sebuah instans / obyek, yang otomatis memberikan informasi tentang obyek yang menjadi pemiliknya.

IV.1.1 Penggunaan Variabel

Salah satu penggunaan variabel adalah dalam keperluan mencetak data, seperti pada baris ke-14 sampai baris ke-24 Contoh 4.1. Misalnya untuk baris ke-13 sampai ke-16 pada Contoh 4.1 :

```
12   ...
13      //menampilkan Nomor Polisi
14      System.out.println("Nomor Polisi : " + noPol);
15      //menampilkan Merk
16      System.out.println("Merk : "+merk);
17   ...
```

nilai variabel *noPol* dan *merk* akan tercetak pada layar monitor sebagai berikut:

Nomor Polisi : D 234 LE
Merk : Suzuki Escudo

IV.1.2 Deklarasi dan Inisialisasi Variabel

Pada contoh 4.1, semua perintah mencetak pada method *tampilkanInfoMobil( )* akan mengambil nilai / data dari variabel-variabel atribut / anggota yang telah didefinisikan di class *Mobil*. Agar nilai dapat diambil, maka harus dideklarasikan terlebih dahulu variabel tempat menyimpan nilai tersebut.

Adapun syntax umum deklarasi dan inisialisasi adalah sebagai berikut :

```
[modifiers] data_type identifier [ = value ] ;
```

di mana :

e. [*modifiers*] merepresentasikan beberapa kata kunci pada teknologi Java, misalnya *public, private, static,* dll.
f. *data_type* merepresentasikan tipe data dari variabel, misalnya *int, String, float,* < nama class >, dll.
g. *identifiers* merepresentasikan nama variabel
h. *value* adalah nilai awal (inisialisasi) dari variabel. Pencantuman *value* ini sifatnya opsional, karena inisialisasi variabel tidak harus dilakukan bersamaan dengan deklarasi.

Contoh 4.2

```
1   public class Mobil{
2
3       //definisi method tampilkanInfoMobil( )
4       public void tampilkanInfoMobil( ){
5           //menampilkan Nomor Polisi
6           System.out.println("Nomor Polisi : " + noPol);
7           //menampilkan Merk
8           System.out.println("Merk : "+merk);
9           //menampilkan Harga
10          System.out.println("Harga : "+harga);
11          //menampilkan Tahun Pembuatan
12          System.out.println("Tahun     Pembuatan      :
            "+tahunPembuatan);
13          //menampilkan Nama Pemilik
```

```
14          System.out.println("Nama Pemilik "+namaPemilik);
15      }//akhir dari method tampilkanInfoMobil
16
17      //main method
18      public static void main ( String[ ] args ){
19      //instanstiasi obyek mobil1
20      Mobil mobil1 = new Mobil ( );
21      //perintah untuk menampilkan data tentang mobil1
22      mobil1.tampilkanInfoMobil( );
23      }
24  }//akhir dari class
```

Misalnya class *Mobil* hanya berisi method *tampilkanInfoMobil( )* saja, seperti pada Contoh 4.2, maka ketika dilakukan *compiling*, maka program akan mengalami kegagalan compile, dan JVM akan mengeluarkan pesan error sebagai berikut:

```
Mobil.java:6: cannot resolve symbol
symbol   : variable noPol
location: class Mobil
System.out.println("Nomor Polisi : " + noPol);
                                     ^
Mobil.java:8: cannot resolve symbol
symbol   : variable merk
location: class Mobil
System.out.println("Merk : "+merk);
                             ^
Mobil.java:10: cannot resolve symbol
symbol   : variable harga
location: class Mobil
System.out.println("Harga : "+harga);
                              ^
Mobil.java:12: cannot resolve symbol
symbol   : variable tahunPembuatan
location: class Mobil
System.out.println("Tahun         Pembuatan          :
"+tahunPembuatan);
 ^
Mobil.java:14: cannot resolve symbol
symbol   : variable namaPemilik
location: class Mobil
System.out.println("Nama Pemilik "+namaPemilik);
                                   ^
5 errors
```

Pesan error *cannot resolve symbol*, berarti sistem tidak mengenali simbol-simbol yang tertentu, yang dalam kasus ini simbol-simbol tersebut adalah variabel-variabel *noPol, merk, harga, tahunPembuatan,* dan *namaPemilik*. Hal ini dikarenakan, JVM tidak menemukan referensi yang menjelaskan jenis simbol (class / variabel /

obyek), dan tipe datanya (primitif / referensi), karena memang simbol-simbol tersebut belum didefinisikan sebelumnya.

Proses pendefinisian variabel dinamakan *deklarasi*. Secara teknis, deklarasi berarti perintah kepada sistem (JVM) untuk mengalokasikan satu blok memori dengan ukuran tertentu (besarnya dalam byte, tergantung dari type datanya).

Contoh deklarasi variabel dapat dilihat pada baris ke-3 dan ke-5 Contoh 4.3. Contoh 4.3 merupakan variasi dari Contoh 4.1 dengan penambahan baris 12b dan 12c. Variabel bernama *noPol* dideklarasikan dengan tipe data *String*. Artinya, JVM akan mengalokasikan satu blok memori yang akan mereferensi ke suatu kumpulan blok memori yang akan menampung data *String* (kumpulan data character).

Pada baris ke-5 Contoh 4.3, sebuah variabel *harga* dideklarasikan dengan tipe data *int*, artinya JVM akan mengalokasikan satu blok memori yang akan digunakan sebagai media penyimpanan nilai untuk variabel *harga*.

Sedangkan inisialisasi variabel adalah pernyataan / statement pemberian nilai awal variabel setelah variabel tersebut dideklarasikan. Secara teknis, inisialisasi variabel berarti memasukkan suatu nilai ke memori yang dialokasikan untuk variabel tersebut.

Pada contoh 4.3, proses inisialisasi variabel dilakukan pada method *tampilkanInfoMobil( )*. Untuk tipe data *String*, nilainya dinyatakan dalam tanda kutip "…".

EKSPERIMEN

1. Buatlah beberapa obyek dari class : "Kucing", "Negara", "Baju", "PersegiPanjang", "Lingkaran". Buatlah variabel-variabel atribut pada masing-masing class. Buatlah method seperti pada Contoh 4.1 :

```
public static void main (String[] args){
        // code block
}
```

   di mana pada code block ditampilkan nilai-nilai dari atribut setiap obyek, mirip seperti yang ditunjukkan pada Contoh 4.1.

Contoh 4.3

```
  2  ...
  3  public String noPol;
     ...
  5  public int harga;
     ...
 12  public void tampilkanInfoMobil( ){
12b  noPol = “D 234 LE”;
12c  harga = 70000000;
     ...
     }
```

Deklarasi dan inisialisasi variabel dapat dilakukan sekaligus. Pada contoh 4.5 diperlihatkan proses deklarasi dilakukan sekaligus dengan memasukkan nilai awal (inisialisasi) ke variabel yang dideklarasikan.

Pada baris ke-3 Contoh 4.5, variabel *noPol* dideklarasikan sebagai variabel tipe *String*, dengan nilai awal adalah “D 234 LE”.

Pada baris ke-5 Contoh 4.5, variabel *harga* dideklarasikan sebagai variabel tipe *int* dengan nilai awal adalah 70000000.

Cara deklarasi dan inisialisasi variabel sekaligus ini dapat diterapkan di luar method. Berbeda dengan cara pertama, di mana inisialisasi yang berada pada *statement* yang terpisah dengan deklarasi harus dilakukan di dalam method.

Contoh 4.5

```
2  ...
3  public String noPol = “D 234 LE”;
   ...
5  public int harga = 70000000;
```

IV.2 Mendeskripsikan Tipe Data Primitif

Dari contoh-contoh 4.1 sampai 4.5 telah ditunjukkan beberapa tipe data, yang terlihat pada saat dilakukan deklarasi. Telah disebutkan bahwa tipe data menunjukkan besarnya alokasi memori yang ditempatkan untuk menyimpan nilai variabel.

Pada modul IV ini akan dibahas tipe data primitif, yaitu tipe data yang digunakan untuk variabel yang nilainya ditempatkan pada alokasi memori yang telah ditentukan. Tipe data yang lain adalah tipe data referensi, yaitu tipe data yang digunakan untuk variabel yang alokasi memorinya memuat referensi ke alamat memori lain, bukan memuat nilai. Nilai sesungguhnya berada di memori lain yang ditunjuk.

Terdapat beberapa jenis tipe data primitif, antara lain :

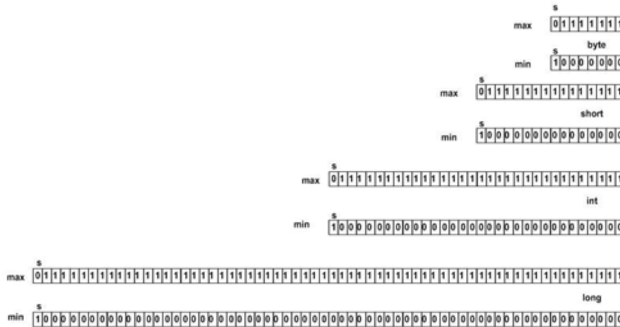
a. **Integral**, merepresentasikan nilai-nilai bilangan bulat (tidak memiliki elemen pecahan desimal.
b. **Floating point**, merepresentasikan nilai-nilai bilangan real (memiliki elemen pecahan desimal)
c. **Tekstual**, merepresentasikan nilai-nilai berupa alphabet.
d. **Logika**, merepresentasikan nilai-nilai logika (hanya bernilai *true* dan *false*).

IV.2.1 Tipe Data Integral

Tipe data integral merupakan tipe data untuk variabel yang nilai-nilainya adalah bilangan bulat (tidak memiliki elemen pecahan desimal). Ada 4 tipe data yang merupakan tipe data integral. Masing-masing diperlihatkan pada Tabel IV.1.

Tabel IV.1.
Tipe Data Integral

Tipe Data	Panjang	Rentang Nilai	Contoh Nilai
byte	8 bit	-2^{7} sampai 2^{7} (-128 sampai 127) (256 kemungkinan nilai)	5 -126
short	16 bit	-2^{15} sampai 2^{15} (-32.768 sampai 32.767) (65.535 kemungkinan nilai)	9 -23659
int	32 bit	-2^{31} sampai 2^{31} (-2.147.483.648 sampai 2.147.483.647) (4.294.967.296 kemungkinan nilai)	2067456397 -1456398567
long	64 bit	-2^{-63} sampai 2^{63} (-9.223.372.036.854.775.808 sampai 9.223.372.036.854.775.807) (18.446.744.073.709.551.616 kemungkinan nilai)	3L -2147483648L 67L



Gambar 4.1. Representasi biner dari Keempat Tipe Data Integral

Pada Tabel IV.1, terlihat bahwa masing-masing tipe data integral mempunyai panjang representasi biner yang berbeda, mulai dari 8 bit (tipe *byte*), sampai 64 bit (tipe *long*). Perbedaan representasi biner ini berpengaruh pada rentang nilainya.

Untuk menyatakan nilai variabel bertipe *long*, perlu ditambahkan huruf 'L' di belakang angka.

Tipe data integral memiliki 1 *sign bit* yang menempati bit dengan urutan tertinggi (byte : bit ke-7, short : bit ke-15, int : bit ke-31, long : bit ke-63). Gambar 4.1 memperlihatkan representasi biner dari setiap tipe data, dengan menampilkan representasi untuk bilangan maksimum dan minimum.

IV.2.2 Tipe Data Floating Point

Tipe data *floating point* merupakan tipe data untuk untuk variabel yang nilai-nilainya adalah bilangan real (dapat mempunyai pecahan desimal). Tabel IV.2 memperlihatkan 2 jenis tipe data floating-point, yaitu *float* dan *double*.

Tabel IV.2
Tipe Data Floating-Point

Tipe Data	Panjang	Contoh Penulisan Nilai yang Diperbolehkan
float	32 bit	78F -34736.86F 6.4E4F (sama dengan 6,4 x 10^4)
double	64 bit	-2356 3.5E7 67564788965.567

IV.2.3 Tipe Data Textual

Tipe data textual merupakan tipe data untuk variabel yang nilai-nilainya adalah karakter tunggal. Tipe data yang merupakan tipe data textual adalah *char* yang memiliki panjang 16 bit. Nilai variabel *char* ditulis dengan diberi tanda kutip tunggal '...'.

Contoh penggunaan tipe data *char* :

```
public char alphabet = 'A';
public char ascii = '\111'; // jika dicetak, akan
                            //menghasilkan
                           // huruf 'I';
```

IV.2.4 Tipe Data Logika

Tipe data logika adalah tipe data yang hanya memiliki 2 kemungkinan nilai, yaitu *true* atau *false*. Hanya satu tipe data logika pada teknologi Java, yaitu *boolean*.

Contoh penggunaan tipe data *boolean* :

```
public boolean status = true;
public boolean check = 10 < 5 ; // nilai check menjadi
                                //false
public boolean hasil = (10<5) && (var==3);
```

IV.2.5 Memilih Tipe Data

Pemilihan tipe data pada saat membuat program merupakan salah satu aspek yang harus diperhatikan. Beberapa hal yang perlu diperhatikan dalam memilih tipe data antara lain :

a. Perhatikan jenis data yang akan digunakan dalam program, apakah berupa bilangan bulat, bilangan real, nilai logika, atau karakter. Sesuaikan tipe data dengan kebutuhan jenis data tersebut.
b. Perhatikan dalam rancangan program, apakah terdapat operasi pembagian. Ketika terdapat operasi pembagian, sangat disarankan menggunakan variabel dari jenis data yang mengakomodasi bilangan real, supaya tidak terjadi *loss of precision*.
c. Untuk program-program yang memperhatikan ukuran memory, seperti aplikasi *mobile phone*, pemilihan data dengan ukuran yang lebih kecil lebih direkomendasikan, misalnya *byte, short, float*.

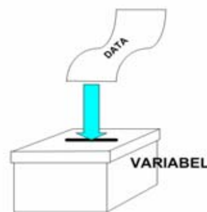
IV.3 Mendeklarasikan Variabel dan Melewatkan Nilai ke Variabel

Variabel adalah satu entitas penyimpanan data yang paling elementer. Dalam pemrograman, variabel memegang peran yang sangat penting, karena suatu program sangat tergantung pada adanya data. Tanpa data, operasi-operasi pada program tidak dapat dijalankan.

Sebagai contoh, jika diinginkan suatu operasi penjumlahan tetapi tidak terdapat 2 buah data yang akan dijumlahkan, maka operasi penjumlahan tidak dapat dilakukan.

Variabel sendiri tidak serta-merta dapat diartikan sebagai data itu sendiri. Variabel, dalam terminologi pemrograman, lebih mengacu kepada alokasi memori yang dapat diisikan data. Karena mengacu kepada alokasi memori, maka data yang dapat diisikan pada variabel, normalnya, dapat berubah. Hubungan antara variabel dan data diilustrasikan pada Gambar 4.2.

Agar data dapat dipergunakan dalam operasi-operasi pada program, maka perlu didefinisikan alokasi memori yang dibutuhkan untuk menyimpan data tersebut. Pendefinisian alokasi memori tersebut dinamakan *deklarasi*.



Gambar 4.2. Variabel Sebagai Tempat Menyimpan Data

IV.3.1 Penamaan Sebuah Variabel

Variabel pada dasarnya adalah alokasi memori. Masalah akan timbul apabila terdapat beberapa alokasi memori yang dapat diisikan data. Data harus dialokasikan ke alokasi memori yang tepat. Oleh karena itu, selain alokasi memori, dibutuhkan juga nama dari alokasi memori tersebut, supaya program dapat melacak lokasi memori yang menyimpan data yang diinginkan.

Penamaan variabel pada dasarnya dibebaskan kepada programmer. Nama yang diberikan kepada variabel disebutt *identifier.* Penamaan identifier variabel pada pemrograman Java mempunyai beberapa aturan, sebagai berikut :

a. Identifier variabel harus dimulai dengan alfabet huruf besar, huruf kecil, tanda dollar ($) atau *underscore* (_). Setelah karakter pertama, dapat diikuti dengan angka.

b. Identifier variabel tidak boleh mengandung *punctuation*, spasi, atau *dashes (-)*
c. Kata kunci pada teknologi Java , seperti pada Tabel IV.3, tidak dapat dijadikan nama identifier variabel.

Tabel IV.3
Kata-kata Kunci pada Teknologi Java yang Tidak Dapat Dijadikan Nama Variabel

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

Contoh nama variabel yang diizinkan :
a. @2var
b. _status
c. tanggal
d. jumlahBarang
e. nama_kecil
f. final_test
g. int_float

IV.3.2 Melewatkan Sebuah Nilai ke Sebuah Variabel

Setelah variabel dideklarasikan dan diberi nama, maka langkah berikutnya adalah melewatkan sebuah nilai ke variabel tersebut. Nilai yang diberikan kepada sebuah variabel haruslah sesuai dengan tipe data variabel tersebut. Misalnya untuk nilai berupa karakter harus dilewatkan ke variabel dengan tipe data *char*, tidak boleh dilewatkan ke variabel dengan tipe data *int*.

Contoh melewatkan nilai ke sebuah variabel adalah pada Contoh 4.1 :

```
1   ...
2         public String noPol = "D 234 LE";
3         public int harga = 70000000;
4
5   ...
```

IV.3.3 Konstanta

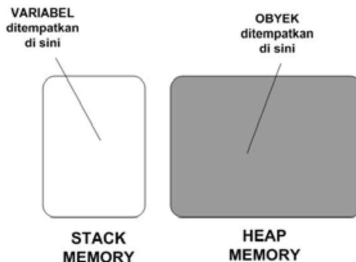
Konstanta adalah variabel yang nilainya tidak dapat diubah-ubah. Dalam aplikasi terkadang dibutuhkan suatu variabel yang dicegah untuk dimodifikasi oleh program lainnya. Hal ini dilakukan untuk menjaga agar logika program tetap terjaga. Seperti misalnya pada class Mobil pada Contoh 4.2 :

```
1   public class Mobil{
2           public final int jumlahRoda=4;
3   }
```

Seperti diperlihatkan, untuk membuat konstanta, digunakan sebuah modifier final.

IV.3.4 Menyimpan Variabel Primitif dan Konstanta pada Memori

Pada teknologi Java, penyimpanan variabel primitif dan konstanta dilakukan pada *Stack Memory*. Panjang memory yang menjadi tempat penyimpanan variabel tergantung dari panjang tipe data dari variabel. Misalnya untuk menyimpan variabel yang bernilai integer, maka diperlukan 32 bit pada stack memory untuk menyimpan data tersebut.



Gambar 4.3 Stack dan Heap Memory

Pada teknologi Java, sebenarnya ada 2 macam memory yang dipergunakan untuk menyimpan variabel, yaitu *Stack Memory* dan *Heap Memory*. Heap memory dipergunakan untuk menyimpan variabel referensi (bukan variabel primitif).

5 MENGGUNAKAN OPERATOR ARITMATIKA DAN BITWISE

V.1 Menggunakan Operator Aritmatika untuk Memodifikasi Nilai

Modifikasi nilai, selain dilakukan menggunakan penugasan (*assignment*), dapat juga dilakukan dengan menggunakan operasi aritmatika. Operasi aritmatika ini menggunakan operator-operator aritmatika. Penggunaan operator aritmatika dalam membentuk pernyataan aritmatika memiliki format berikut :

```
<variabel penyimpan hasil akhir> =
        <operasi aritmatika> ;
```

Jadi ruas kiri tanda "=" merupakan nama variabel yang akan digunakan sebagai tempat menyimpan hasil dari operasi aritmatika pada ruas kanan. Penulisannya tidak dapat dilakukan sebaliknya.

Contoh penulisan operasi aritmatika yang benar :

```
a = b + 5;
```

Contoh penulisan operasi aritmatika yang salah :

```
b+5 = a;
```

Operator aritmatika ada yang merupakan operator binary (membutuhkan 2 buah operand) dan operator unary (membutuhkan 1 buah operand).

Operator aritmatika yang menggunakan 2 buah operand (binary) dapat dilihat pada Tabel 5.1.

Tabel 5.1
Operator Aritmatika Binary

Arti Operator	Operator	Contoh Pemakaian	Keterangan
Penjumlahan	+	sum=num1 + num2	
Pengurangan	-	diff=num1 - num2	
Perkalian	*	prod=num1 * num2	
Pembagian	/	quot=num1 / num2	jika num1 dan num2 adalah integer, pembagian akan menghasilkan nilai integer tanpa mengikutsertakan sisa, jika terdapat sisa.
Sisa (modulus)	%	mod=num1 % num2	Hasil operasi modulus adalah sisa dari operasi num1 / num2. Hasil operasi modulus memiliki tanda (+/-) yang sama dengan operand pertama

Sedangkan operator aritmatika yang membutuhkan 1 buah operator (unary), dapat dilihat pada Tabel 5.2.

Tabel 5.2.
Operator Aritmatika Unary

Arti Operator	Operator	Contoh Pemakaian	Keterangan
Pre-Increment	++operand	int i = 8 ; int j = ++i; i bernilai 8, j bernilai 8	

Post-Increment	operand++	int i = 8;
		int j = i++;
		i bernilai 9, j bernilai 8	

Arti Operator	Operator	Contoh Pemakaian	Keterangan
Pre-Decrement	--operand	int i = 8 ; int j = --i; i bernilai 7 , j bernilai 7	
Post-Increment	operand--	int i=8; int j = i--; i bernilai 7, j bernilai 8	

Ada konsep mendasar tentang operator *post* dan *pre*, seperti yang ditunjukkan pada Contoh 5.1.

Contoh 5.1

```
1  int a = 10;
2  int b = a++; // nilai b = 10, nilai a = 11
3  int c = ++a; // nilai c = 12, nilai a = 12
```

Pada post-increment, baris ke-2, nilai awal variabel a di-*copy* ke variabel b. Kemudian nilai variabel a ditambah 1, sehingga nilai terakhir b adalah 10, dan nilai terakhir a adalah 11.

Kemudian pada pre-increment, baris ke-3, nilai variabel a ditambah 1, kemudian hasilnya di-*copy* ke variabel b, sehingga nilai terakhir b adalah 12, dan nilai terakhir a juga 12.

Urutan proses yang sama juga berlaku pada post-decrement dan pre-decrement.

V.2 Menggunakan Operator Bitwise untuk Memodifikasi Nilai

Operator bitwise adalah operator yang digunakan untuk mengubah nilai suatu variabel dengan cara melakukan manipulasi pada bit. Operator-operator tersebut dapat dilihat pada Tabel 5.3.

Tabel 5.3.
Operator Bitwise

Arti Operator	Operator	Contoh Pemakaian	Keterangan

Shift Kiri	<<	int b = -16;
		int c = b<<2;
		nilai c = -64	

Shift Kanan, dengan pengisian "0" pada bit-bit sebelah kiri	>>>	int b = -16;
		int c = -16>>>2;
		nilai c = 1073741820
			Ketika digeser ke kanan 1 kali, maka bit paling kiri terisi dengan 0

AND	&	int a = 12;
		int b = - 13;
		int c = a & b;
		nilai c = 0
			Yang di-AND adalah setiap bit dari a dan b yang menempati posisi bit yang sama, misalnya bit ke-2 variabel a di-AND dengan bit ke-2 variabel b.

OR	|	int a = 12;
		int b = -13;
		int c = a | b;
		nilai c = -1
			Yang di-OR adalah setiap bit dari a dan b yang menempati posisi bit yang sama

Exclusive OR	^	int a = 13;
		int b = -13;
		int c = a ^ b ;
		nilai c = -2
			Yang di-exclusive OR adalah setiap bit dari a dan b yang menempati posisi bit yang sama

Complement	~	int a = 12;
		int c = ~a;
		nilai c = -13
			Nilai setiap bit diganti dengan lawannya. Jika bit bernilai 1, maka nilai tersebut akan dirubah menjadi 0

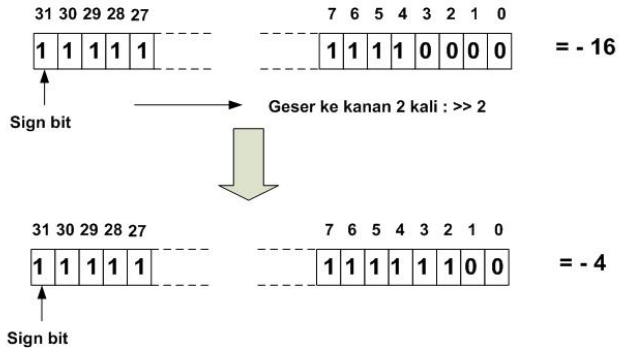
Contoh penggunaan operator bitwise adalah sebagai berikut :

a. Shift kanan

Contoh 5.2.

```
int b = -16;
int c = b>>2;
// nilai c = -4
```

Operator ini akan menggeser bit-bit pada variabel b sebanyak 2 langkah ke kanan, dengan membuang 2 bit pada lokasi bit dengan nomor index paling kecil, dan menambahkan 2 bit dengan nilai yang sama dengan *sign* bit pada 2 lokasi bit dengan nomor index paling besar. Visualisasi proses ini dapat dilihat pada Gambar 5.1.



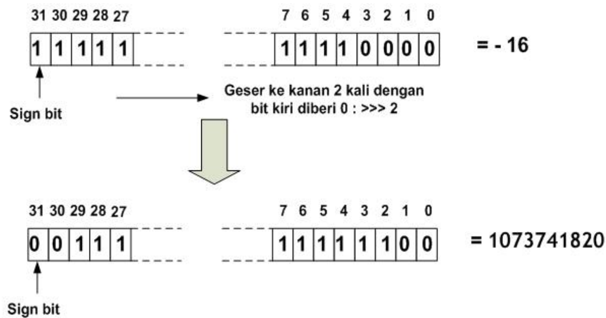
Gambar 5.1 Shift Kanan

b. Shift kanan dengan bit kiri diberi nilai bit '0'

Contoh 5.3.

```
int b = -16;
int c = -16>>>2;
// nilai c = 1073741820
```

Operator ini akan menggeser bit-bit pada variabel b sebanyak 2 langkah ke kanan, dengan membuang 2 bit pada lokasi bit dengan nomor index paling kecil, dan menambahkan bit '0' pada 2 lokasi bit dengan nomor bit paling besar. Visualisasi proses ini dapat dilihat pada Gambar 5.2



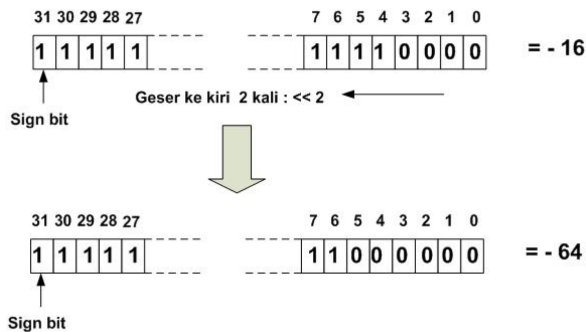
Gambar 5.2 Shift Kanan dengan Bit Kiri diberi Nilai Bit '0'

c. Shift Kiri

Contoh 5.4.

```
int b = -16;
int c = b<<2;
// nilai c = -64
```

Operator ini akan menggeser bit-bit pada variabel b sebanyak 2 langkah ke kiri, dengan membuang 2 bit pada lokasi bit dengan nomor index paling besar, dan menambahkan '0' pada 2 lokasi bit dengan nomor bit paling kecil. Visualisasi proses ini dapat dilihat pada Gambar 5.3.



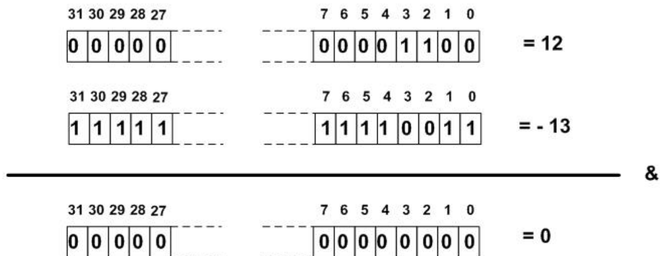
Gambar 5.3. Shift Kiri

d. Operator ‘&’

Contoh 5.5.

```
int a = 12;
int b = - 13;
int c = a & b;
//nilai c = 0
```

Operator ini akan membandingkan nilai bit milik dua variabel dengan nomor index yang bersesuaian menggunakan operator AND (‘&’), di mana nilai 1 akan didapatkan jika kedua bit yang dibandingkan bernilai 1. Visualisasi proses ini dapat dilihat pada Gambar 5.4.



Gambar 5.4. Contoh Penggunaan Operator ‘&’

e. Operator ‘|’

Contoh 5.6.

```
int a = 12;
int b = -13;
int c = a | b;
// nilai c = -1
```

Operator ini akan membandingkan nilai bit milik dua variabel dengan nomor index yang bersesuaian menggunakan operator OR (‘|’), di mana nilai 1 akan didapatkan jika salah satu dari kedua bit yang dibandingkan bernilai ‘1’. Visualisasi proses ini diperlihatkan pada Gambar 5.5.

Gambar 5.5. Contoh Penggunaan Operator '|'

f. Operator '^'

Contoh 5.7.

```
int a = 13;
int b = -13;
int c = a ^ b ;
// nilai c  = -2
```

Operator ini akan membandingkan nilai bit milik dua variabel dengan nomor index yang bersesuaian menggunakan operator Exclusive OR (^), dimana nilai '1' akan didapatkan jika nilai bit kedua variabel berbeda. Visualisasi proses ini diperlihatkan pada Gambar 5.6.

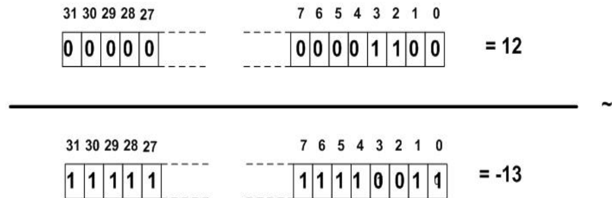
Gambar 5.6. Contoh Penggunaan Operator '^'

g. Operator '~'

Contoh 5.8.

```
int  a = 12;
int  c = ~a;
//nilai c = -13
```

Operator ini akan melakukan komplemen pada setiap bit dari variabel a, artinya jika bit pada suatu lokasi bernilai '0', maka nilai bit tersebut diubah menjadi '1', dan jika nilai bit bernilai '1', maka nilai bit tersebut akan diubah menjadi '0'. Visualisasi proses ini dapat dilihat pada Gambar 5.7.



Gambar 5.7. Contoh Penggunaan Operator '~'

EKSPERIMEN

1. Buatlah program sebagai berikut :

```
public class Test() {
        public static void main (String[] args ){
             int a = -60;
             int b = a>>1;
             int c = a>>2;
             int d = a>>3;
             int e = a>>4;
             int f = a>>5;
             System.out.println(b + " " + c + " " + d +
                                   " " + e + " " + f);

        }
}
```

Perhatikan nilai-nilai yang dihasilkan oleh b, c, d, e, dan f. Jawablah : Carilah fungsi matematika yang mempunyai sifat mirip dengan perilaku operator >> pada bilangan negatif !

2. Buatlah program sebagai berikut :

```
public class Test() {
        public static void main (String[] args ){
             int a = 60;
             int b = a>>1;
             int c = a>>2;
             int d = a>>3;
             int e = a>>4;
             int f = a>>5;
             System.out.println(b + " " + c + " " + d +
                                      " " + e +" " + f);
        }
}
```

Perhatikan nilai-nilai yang dihasilkan oleh b, c, d, e, dan f. Jawablah : Carilah fungsi matematika yang mempunyai sifat mirip dengan perilaku operator >> pada bilangan positif !

3. Buatlah program sebagai berikut :

```
public class Test() {
        public static void main (String[] args ){
             int a = 60;
             int b = a<<1;
             int c = a<<2;
             int d = a<<3;
             int e = a<<4;
             int f = a<<5;
             System.out.println(b + " " + c + " " + d +
                                       " " + e + " " + f);
        }
}
```

Perhatikan nilai-nilai yang dihasilkan oleh b, c, d, e, dan f. Jawablah : Carilah fungsi matematika yang mempunyai sifat mirip dengan perilaku operator << pada bilangan positif !

V.3 Prioritas Operator

Jika pada suatu operasi aritmatika atau bitwise terdapat lebih dari satu operator, maka compiler akan melakukan prioritisasi perhitungan. Prioritas yang diberikan oleh compiler dalam melakukan perhitungan adalah sebagai berikut (dimulai dari prioritas tertinggi) :

a. Operator yang berada dalam tanda kurung "(...)" atau disebut juga *parantheses*.
b. Operator-operator increment atau decrement
c. Operator - operator perkalian atau pembagian, yang urutan operasinya dari kiri ke kanan.
d. Operator-operator penjumlahan atau pengurangan, yang urutan operasinya dari kiri ke kanan.
e. Operator bitwise, dengan urutan operasi dari kiri ke kanan, dan dimulai dari operator bitwise paling kiri diikuti operator di sebelah kanannya dan seterusnya.

Contoh prioritisasi operator diperlihatkan pada Contoh V.9.

Contoh 5.9

```
int c = 12 * 3 +5 / (8 - 3) ;
```

Maka urutan operasinya adalah sebagai berikut :

```
int c = 12 * 3 + 5 / 5 ;
int c = 36 + 5 / 5;
int c = 36 + 1 ;
int c = 37;
```

Contoh lain melibatkan operator bitwise, seperti pada Contoh 5.10.

Contoh 5.10.

```
int c = 3 + 4 >> 1 + 1 << 1;
```

Maka urutan operasinya adalah sebagai berikut :

```
int c = 7 >> 1 + 1 << 1;
int c = 7 >> 2 << 1;
int c = 1 << 1;
int c = 2;
```

EKSPERIMEN

Analisa hasil operasi aritmatika berikut ini dan buktikan jawaban Anda dengan membuat program :

a. x = (2 + 12) / 7 - 2
b. x = 4.0 >> 2.0
c. x = 3.0 <<1

Apakah kesimpulan Anda dari eksperimen ini ?

V.4 Menggunakan Promosi dan *Type Casting*

Promosi dan *Type Casting* adalah fitur-fitur pada teknologi Java yang berfungsi mengubah representasi bit dari variabel-variabel primitif (selain *boolean*).

V.4.1 Promosi

Promosi adalah proses pengubahan representasi bit variabel primitif dari representasi bit yang lebih rendah ke representasi bit yang lebih tinggi. Promosi dapat terjadi apabila :

a. Jika terjadi *assigning* nilai dari tipe data dengan representasi bit yang lebih kecil ke tipe data dengan representasi bit yang lebih besar, seperti yang diperlihatkan pada Contoh 5.11.

Contoh 5.11.

```
short a = 12;
int b = a ;
```

b. Jika terjadi *assigning* nilai dari tipe data integral ke tipe data floating-point, seperti yang diperlihatkan pada Contoh 5.12.

Contoh 5.12.

```
int a = 30;
float b = a ;
```

V.4.2 Type Casting

Type casting merupakan proses pengubahan representasi bit variabel primitif dari representasi bit yang lebih tinggi ke representasi bit yang lebih rendah.

Syntax dari *type casting* adalah sebagai berikut :

```
identifier = (target_type) value ;
```

di mana :

a. *identifier* = nama variabel yang menjadi tempat penyimpanan nilai
b. *target_type* = tipe data yang diinginkan menjadi tipe data dari *value*.
c. *value* = *nilai* yang akan di-*casting*.

Penggunaan *type casting* diperlihatkan pada Contoh5.13.

Contoh 5.13.

```
int num1 = 34;
int num2 = 45;
short num3 = (short)(num1 + num2 );
System.out.println(num3);
```

Penjumlahan variabel num1 dan num2 akan menghasilkan nilai *integer*. Nilai integer ini tidak dapat di-*assign* di variabel num3 yang bertipe data *short*. Setelah dikompilasi dan dieksekusi, maka program akan menghasilkan *output* yaitu :

```
79
```

Contoh 5.14 menunjukkan apabila pada Contoh 5.13 tidak dilakukan *casting*.

Contoh 5.14.

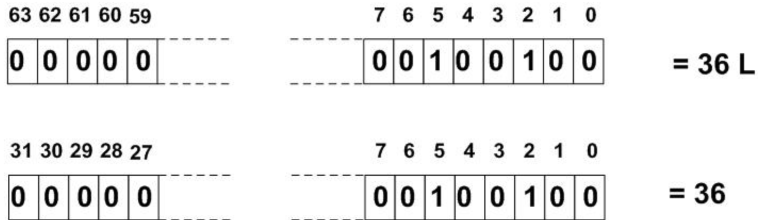
```
int num1 = 34;
int num2 = 45;
short num3 = (num1 + num2 );
System.out.println(num3);
```

Maka ketika program dikompilasi, kompilasi akan gagal, dan compiler akan memberikan pesan error :

```
Test.java:20: possible loss of precision
found   : int
required: short
short num3 = (num1+num2);
                  ^
1 error
```

Type casting, sebenarnya adalah proses “pemotongan” / *chopping* bit. Bit yang dipotong adalah bit-bit dengan nomor index

paling besar. Bit-bit tersebut dipotong sehingga didapatkan representasi bit yang diinginkan. Visualisasi *type casting* diperlihatkan pada Gambar 5.8.



Gambar 5.8. Type Casting

Pada Gambar 5.8 diperlihatkan representasi angka 36 yang bertipe data *long* (64 bit). Representasi ini akan dibuat menjadi representasi *int* (32 bit). Cara melakukan *casting* adalah dengan memotong bit ke-32 sampai ke-63 dari representasi *long*, sehingga hanya tersisa bit ke-0 sampai ke-31. Meskipun demikian, karena nilainya masih di dalam batas integer, maka nilainya tetap 36.

Perhatikan Contoh 5.15. Contoh 5.15 memperlihatkan suatu variabel bertipe-data *long* yang nilainya berada di luar batas tipe data *int*. Variabel *long* ini akan di-*casting* ke *int*.

Contoh 5.15.

```
int num1;
long num2 = 123987654321L;
num1 = (int)(num2);
System.out.println(num1);
```

Ketika program dikompilasi dan dieksekusi, maka program akan mengeluarkan *output* sebagai berikut :

```
-566397263
```

Nilai variabel *num1* yang berbeda dari *num2* disebabkan oleh dipotongnya bit ke-32 sampai ke-63 dari variabel num2.

EKSPERIMEN

1. Analisa program berikut ini. Buktikan hasil analisa Anda dengan keluaran yang dihasilkan :
   a. float x = (float)(int)(double)(short)3;
   b. float x = (double)(int)(double)(short)3;
   c. int y = (int)4.5;
   d. short b = (byte)128<<1;

   Buat kesimpulan dari eksperimen ini.

2. Bandingkan dua operasi aritmatika berikut ini :
   a. byte bt = 135;
   b. byte bt = (byte)135;

   Buat program yang menjalankan kedua operasi aritmatika tersebut. Apakah hasilnya berbeda ? Jika berbeda, mengapa ?

V.4.3 Beberapa Catatan pada Promosi dan *Type Casting*

V.4.3.1 Operasi Aritmatika Menghasilkan Nilai di Luar Batas Tipe Data

Apabila terdapat operasi aritmatika yang hasilnya melewati batas limit tipe data, dan hasilnya disimpan ke variabel dengan tipe data yang batas nilainya di bawah hasil operasi aritmatika tersebut, maka ketika dilakukan kompilasi dan eksekusi, maka hasil operasi aritmatika tersebut adalah negatif. Perhatikan Contoh 5.16.

Contoh 5.16.

```
int a = 55555;
int  b =  66666;
int  c = a * b;
System.out.println(c);
```

Setelah dikompilasi dan dieksekusi, maka program akan menghasilkan:

```
-591337666
```

Hasil ini tidak sesuai dengan yang diinginkan, yaitu 3703629630.

Jika operasi aritmatika yang akan dibuat memiliki kemungkinan menghasilkan nilai di luar batas nilai tipe data, maka sebaiknya pada salah satu operand, tipe datanya dimodifikasi menjadi tipe data dengan representasi bit yang lebih tinggi, sehingga nilai yang akan dihasilkan masih berada di dalam rentang nilai tipe data, dan variabel penyimpan hasil akhirnya juga dimodifikasi ber-tipe data dengan representasi bit yang lebih tinggi. Perhatikan Contoh 5.17.

Contoh 5.17

```
int a = 55555;
long  b =  66666;
long  c = a * b;
System.out.println(c);
```

Setelah dikompilasi dan dieksekusi, maka program akan menghasilkan:

```
3703629630
```

Dengan memodifikasi tipe data operand *b* menjadi *long*, maka operasi aritmatika akan menghasilkan nilai bertipe *long*. Variabel penyimpan data, yaitu *c* dengan demikian juga harus dimodifikasi menjadi *long*.

V.4.3.2 Asumsi Dasar Compiler

Compiler mempunyai asumsi dasar tentang tipe data integral dan floating-point.

Untuk tipe data integral, asumsi-asumsi dasar compiler adalah sebagai berikut :

a. Nilai yang di-*assign* tanpa penambahan keterangan apapun, diasumsikan sebagai nilai *integer*. Perhatikan Contoh 5.18.

Contoh 5.18

```
int a = 12345;
long  b = 3456;
short c = 12367;
```

Pada Contoh 5.18, nilai-nilai yang di-*assign* ke setiap variabel a, b, dan c diasumsikan sebagai *integer*. Pada baris ke-1, nilai 12345 adalah integer, kemudia nilai tersebut di-*assign* ke variabel a.

Begitu juga dengan baris ke-2, nilai 3456 diasumsikan sebagai *integer*. Nilai *integer* tersebut di-*assign* ke variabel b yang bertipe data *long*.

Pada baris ke-3, nilai 12367 diasumsikan sebagai *integer*. Nilai *integer* tersebut di-*assign* ke variabel c yang bertipe data *short.* Meskipun nilai *integer* akan dipotong menjadi 16 bit, tetapi karena nilai 12367 masih berada di dalam batas nilai *short*, maka kompilasi program akan sukses.

Jika ingin keluar dari asumsi, untuk nilai *long*, penulisannya diakhiri dengan huruf "L" atau "l". Perhatikan Contoh 5.19.

Contoh 5.19

```
long  b = 3456L;
```

Jika ingin keluar dari asumsi, untuk nilai *short* dan *byte*, maka dilakukan *type-casting*. Perhatikan Contoh 5.20.

Contoh 5.20.

```
short c = (short)12367;
byte d = (byte)12;
```

b. Jika pada suatu operasi aritmatika, operand-operand pada ruas kanan berbeda tipe datanya (semuanya masih termasuk tipe data integral), dan semua tipe data merupakan tipe data yang representasi bitnya di bawah *integer*, maka hasil operasi aritmatika tersebut akan diasumsikan sebagai integer. Perhatikan Contoh 5.21.

Contoh 5.21.

```
byte theByte = (byte)127 + (short)12346;
```

Jika program ini dikompilasi, maka akan kompilasi akan gagal, dan kompiler akan mengeluarkan pesan error:

```
Test.java:24: possible loss of precision
found   : int
required: byte
byte theByte = (byte)127 + (short)12346;
                          ^
1 error
```

Perhatikan bahwa kompiler menemukan nilai *int* sebagai hasil dari operasi penjumlahan. Ini berarti, hasil penjumlahan tidak dianggap sebagai *short,* tetapi sebagai *int*.

c. Jika pada suatu operasi aritmatika, operand-operand pada ruas kanan berbeda tipe datanya (semuanya masih termasuk tipe data integral), dan salah satu tipe data merupakan tipe data yang representasi bitnya di atas *integer*, yaitu *long*, maka hasil operasi aritmatika tersebut akan diasumsikan sebagai *long*. Perhatikan Contoh 5.22.

Contoh 5.22.

```
byte theByte = (byte)127 + (long)12346;
```

Jika program ini dikompilasi, maka akan kompilasi akan gagal, dan kompiler akan mengeluarkan pesan error:

```
Test.java:24: possible loss of precision
found   : long
required: byte
byte theByte = (byte)127 + (long)12346;
                           ^
1 error
```

Perhatikan bahwa kompiler menemukan nilai *long* sebagai hasil dari operasi penjumlahan. Ini berarti, hasil penjumlahan tidak dianggap sebagai *int,* tetapi sebagai *long*.

Untuk tipe data floating-point, kompiler memiliki asumsi-asumsi sebagai berikut :

a. Nilai yang di-*assign* tanpa penambahan keterangan apapun akan diasumsikan sebagai *double*. Perhatikan contoh 5.23.

Contoh 5.23.

```
double variable = 34.5;
```

Jika diinginkan nilai bilangan real tidak direpresentasikan dalam *double* (atau ingin direpresentasikan dalam *float*), maka penulisan nilainya diakhiri dengan huruf "F" atau "f". Perhatikan Contoh 5.24.

Contoh 5.24.

```
float variable1 = 34.5F;
float variable2 = 3.67f;
```

b. Jika pada operasi aritmatika dengan semua operand pada ruas kanan menggunakan tipe data floating-point, maka hasil operasi aritmatika tersebut direpresentasikan dengan tipe data yang mengikuti tipe data dengan representasi tertinggi pada ruas kanan operasi aritmatika tersebut. Perhatikan Contoh 5.25.

Contoh 5.25.

```
float variable1 = 35.7F + 3.0 ;
```

Contoh 5.25 menggambarkan operasi aritmatika, dengan operand pertama bertipe data *float*, dan operand kedua bertipe data *double.* Hasil operasi aritmatika tersebut disimpan pada *variable1*.
Jika program dikompilasi, maka compiler akan mengeluarkan pesan error sebagai berikut :

```
Test.java:27: possible loss of precision
found   : double
required: float
    float f = 35.7F + 3.0;
                    ^
1 error
```

Pada Contoh 5.25, compiler menganggap hasil penjumlahan merupakan nilai yang bertipe *double*, karena salah satu operand-nya bertipe-data *double*. Dari pesan error ini terlihat bahwa untuk operasi aritmatika dengan salah satu operand-nya bertipe *double*, maka hasil operasi tersebut akan bertipe *double*.

c. Jika pada operasi aritmatika dengan sebagian operand pada ruas kanan menggunakan tipe data floating-point dan sebagian menggunakan tipe data integral, maka hasil operasi aritmatika tersebut direpresentasikan dengan tipe data floating-point yang mengikuti tipe data floating-point dengan representasi bit tertinggi pada ruas kanan operasi aritmatika tersebut. Perhatikan Contoh 5.26.

Contoh 5.26.

```
float variable2 = 35 + 3.0 ;
```

Contoh 5.26 menggambarkan operasi aritmatika, dengan operand pertama bertipe data *integer*, dan operand kedua bertipe data *double.* Hasil operasi aritmatika tersebut disimpan pada *variable2*.
Jika program dikompilasi, maka compiler akan mengeluarkan pesan error sebagai berikut :

```
Test.java:27: possible loss of precision
found    : double
required: float
     float variable2 = 35 + 3.0 ;
                          ^
1 error
```

Pada Contoh 5.26, compiler menganggap hasil penjumlahan merupakan nilai yang bertipe *double*, karena salah satu operand-nya adalah floating-point dan bertipe-data *double*. Dari pesan error ini terlihat bahwa untuk operasi aritmatika dengan salah satu operand-nya bertipe *double* dan operand lainnya bertipe integer, maka hasil operasi tersebut akan bertipe *double*.

EKSPERIMEN

Analisa program berikut ini. Buktikan hasil analisa Anda dengan keluaran yang dihasilkan :

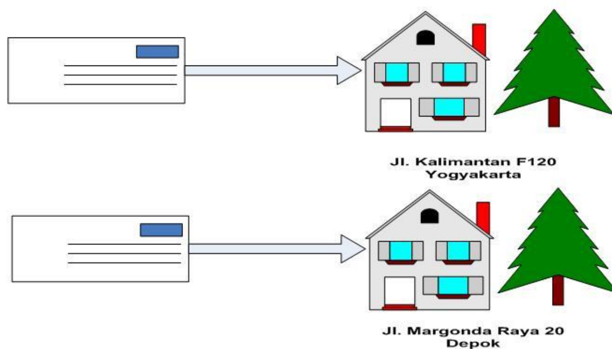
a. int a = (short)20 + 35;
b. float var = 20d + 34.56;

6 MEMBUAT DAN MENGGUNAKAN OBYEK

VI.1 Mendeklarasikan Referensi Obyek, Instantiasi Obyek, dan Inisialisasi Referensi Obyek

Obyek pada dasarnya sama seperti variabel, yang merupakan elemen penyimpanan data, hanya saja ketika mendeklarasikan sebuah obyek, yang dialokasikan pada Stack Memory adalah memori yang tidak akan diisi dengan nilai yang sebenarnya, tetapi diisi dengan alamat memori lain yang berada pada Heap Memory, dengan kata lain memori yang dialokasikan untuk sebuah obyek akan mereferensi / menunjuk ke memori lain pada Heap Memory. Alokasi memory pada Heap Memory adalah alokasi memory untuk menyimpan obyek. Variabel yang menunjuk ke obyek pada Heap Mempory disebut **variabel referensi obyek**.

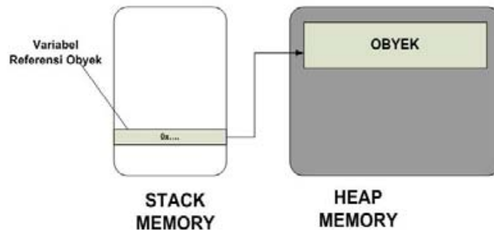
Bayangkanlah variabel referensi obyek seperti sebuah surat yang dialamatkan ke sebuah rumah. Alamat yang tertera pada surat sebenarnya menunjuk kepada rumah tertentu. Obyeknya sendiri adalah rumah dengan alamat yang sama dengan alamat yang tertera pada surat. Gambar 6.1 mengilustrasikan surat yang menunjuk ke rumah yang alamatnya tertera pada surat tersebut.



Gambar 6.1 Satu Surat Menunjuk pada Satu Rumah

Gambar 6.1 memperlihatkan ada 2 buah surat yang akan dikirim ke 2 rumah yang masing-masing memiliki alamat yang berbeda. Surat dapat dikatakan sebagai "perwakilan" dari rumah. Rumah yang diwakilinya dapat diidentifikasi dari alamat rumah yang tertera pada surat.

Secara teknis, variabel referensi obyek ditempatkan pada Stack Memory. Variabel ini menunjuk ke obyek yang berada pada Heap Memory. Ilustrasinya dapat dilihat pada Gambar 6.2.



Gambar 6.2 Lokasi Variabel Referensi Obyek dan Obyek

Obyek pada dasarnya adalah kumpulan variabel. Variabel-variabel di dalam sebuah obyek dapat berupa variabel primitif maupun variabel referensi obyek. Komposisi variabel di dalam obyek mengikuti komposisi class yang menjadi acuan bangunan obyek tersebut.

VI.1.1 Mendeklarasikan Variabel Referensi Obyek

Sama seperti variabel bertipe-data primitif, sebuah variabel referensi obyek harus dideklarasikan terlebih dahulu sebelum dapat digunakan untuk menyimpan nilai.

Bentuk syntax untuk mendeklarasikan variabel referensi obyek adalah :

```
Classname identifier ;
```

di mana :

a. *ClassName* merepresentasikan nama class atau tipe dari obyek yang direferensi oleh referensi obyek.
b. *identifier* merepresentasikan nama variabel.

Contoh 6.1 memperlihatkan class *Rumah* yang menyimpan data berupa *luasTanah, luasBanguan,* dan *harga*. Contoh 6.1 juga memperlihatkan class *AplikasiMakelarRumah* yang tidak terdiri atas variabel. Pada class *AplikasiMakelarRumah* didefinisikan method yang bernama *main ( )*, yang merupakan method utama dalam program yang memuat alur program utama.

Contoh 6.1.

```
1   //File : Rumah.java
2   public class Rumah{
3           public int luasTanah = 100;
4           public int luasBangunan;
5           public int harga;
6           public cetakInfoRumah( ){
7               System.out.println("Luas Tanah = " +
8               luasTanah);
9               System.out.println("Harga = " + harga);
10          }
11  }
```

```
1   //File : AplikasiMakelarRumah.java
2   public class AplikasiMakelarRumah{
3           public static void main(String [] args ){
4                   Rumah rumah1;
5                   Rumah rumah2;
6           }
7   }
```

Pada class *AplikasiMakelarRumah*, baris ke-4 adalah deklarasi variabel referensi obyek yang diberi nama *rumah1*. Baris ke-5 adalah deklarasi variabel referensi obyek yang diberi nama *rumah2*. Kedua variabel tersebut belum menunjuk ke satu obyek pun pada Heap Memory.

VI.1.2 Inisialisasi Variabel Referensi Obyek

Variabel referensi obyek telah dideklarasikan, tetapi obyek secara riil belum ada pada Heap Memory. Pada kondisi ini, variabel referensi obyek memiliki nilai *null*, sehingga belum dapat digunakan untuk memanipulasi data.

Agar dapat memanipulasi data (yang merupakan milik dari obyek), maka perlu diinisialisasi terlebih dahulu obyeknya. Proses inisialisasi obyek disebut juga *instanstiasi obyek*.

Syntax untuk menginisialisasi obyek adalah sebagai berikut :

```
Classname identifier = new Classname ( ) ;
```

atau

```
Classname identifier ;
identifier = new Classname ( );
```

di mana :

a. *Classname* = nama class yang menjadi acuan bangunan obyek.
b. *identifier* = nama variabel referensi obyek
c. *new* = kata kunci pada teknologi Java yang menandakan bahwa obyek
diinstantiasi.

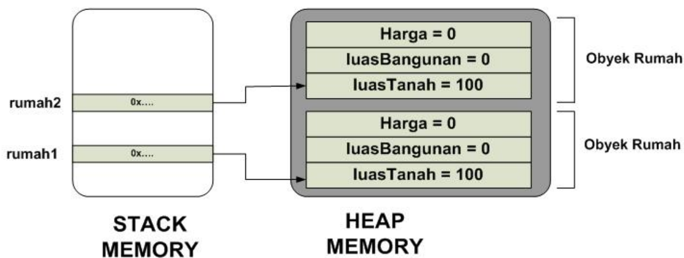
Catatan : untuk sementara, syntax untuk menginisialisasi obyek adalah seperti yang telah ditunjukkan. Pada bab-bab berikut akan dibahas mengenai cara instantiasi obyek yang lebih kompleks.

Contoh 6.2 memperlihatkan class *AplikasiMakelarRumah* dari Contoh 6.1 yang dimodifikasi. Pada baris ke-6, sebuah obyek *Rumah* yang direferensi oleh *rumah1* diinstantiasi, dan pada baris ke-7, sebuah obyek *Rumah* yang direferensi oleh *rumah2* diinstantiasi.

Contoh 6.2.

```
1   //File : AplikasiMakelarRumah.java
2   public class AplikasiMakelarRumah{
3          public static void main(String [] args ){
4                 Rumah rumah1;
5                 Rumah rumah2;
6                 rumah1 = new Rumah( );
7                 rumah2 = new Rumah( );
8          }
9   }
```

Pada class *Rumah,* nilai *luasTanah* ditetapkan = 100, sedangkan *luasBangunan* dan *harga* tidak diberi nilai awal. Ini berarti ketika sebuah obyek *Rumah* diinstantiasi, nilai awal *luasTanah*-nya = 100, *luasBangunan* = 0, dan *harga* = 0. Hasil dari instantiasi pada Contoh 6.2 dapat divisualisasikan pada Gambar 6.3.



Gambar 6.3. Obyek-obyek Rumah yang Telah Diinisialisasi

Gambar 6.3 memperlihatkan bahwa variabel referensi obyek *rumah1* dan *rumah2* berada pada Stack Memory dan menyimpan nilai berupa alamat memori yang ditempati oleh obyek Rumah yang bersesuaian. Nilai awal masing-masing variabel pada setiap obyek sama : *luasTanah = 100, luasBangunan = 0*, *harga = 0*.

EKSPERIMEN

Ubahlah nilai *luasBangunan* pada class *Rumah* pada Contoh 6.1. menjadi 120. Lalu tambahkan baris program untuk menampilkan nilai variabel atribut milik *rumah1* dan *rumah2*. Kompilasilah, lalu jalankan program *AplikasiMakelarRumah*.Perhatikanlah nilai *luasBangunan* setiap obyek Rumah ! Apakah perubahan yang terjadi dibandingkan keluaran / output pada Contoh 6.2 ? Mengapa demikian ?

VI.1.3 Menggunakan Variabel Referensi Obyek untuk Memanipulasi Data

Untuk dapat memanipulasi data pada obyek tertentu, dapat digunakan variabel yang memiliki referensi ke obyek tersebut. Pada baris ke-8 Contoh 6.3 diperlihatkan cara memanipulasi variabel *luasTanah* milik obyek *rumah1*. Demikian pula dengan baris ke-9, diperlihatkan cara memanipulasi variabel *luasBangunan* milik obyek *rumah2*.

Secara umum, syntax untuk mengubah nilai variabel (atau secara umum: mengakses variabel) dari sebuah obyek adalah sebagai berikut :

```
object_name . identifier = < value >;
```

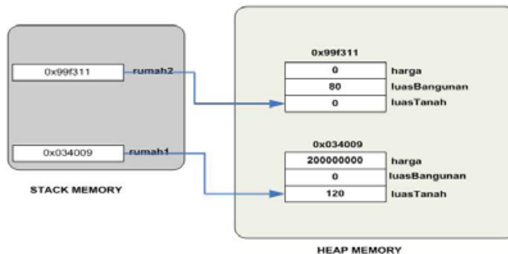
di mana :

a. *object_name* adalah nama obyek yang variabel anggota / atributnya akan diakses / dimanipulasi.
b. *identifier* adalah nama variabel milik *object_name* yang akan diakses / dimanipulasi.
c. *value* adalah nilai yang akan di-*assign* ke variabel.

Contoh 6.3

```
 1   //File : AplikasiMakelarRumah.java
 2   public class AplikasiMakelarRumah{
 3          public static void main(String [] args ){
 4                 Rumah rumah1;
 5                 Rumah rumah2;
 6                 rumah1 = new Rumah( );
 7                 rumah2 = new Rumah( );
 8                 rumah1.luasTanah = 120;
 9                 rumah2.luasBangunan = 80;
10                 rumah1. harga = 200000000;
11
12          }
13   }
```

Visualisasi hasil akhir program pada memory ditunjukkan pada Gambar 6.4.



Gambar 6.4. Menyimpan Variabel Referensi Obyek pada Memory

EKSPERIMEN

1. Manipulasi data-data sebagai berikut :
   a. Buatlah harga rumah2 = 350000000
   b. Buatlah luas bangunan rumah1 = 60
   c. Buatlah luas tanah rumah2 = rumah1, tidak dengan meng-assign luas tanah rumah2 dengan 120, tetapi dengan meng-*assign* luas tanah rumah 2 dengan nilai luas tanah rumah1

2. Buatlah baris-baris program untuk menampilkan semua data pada rumah1 dan rumah2.

VI.1.4 Memindahkan Sebuah Referensi dari Satu Obyek ke Obyek yang Lain

Sebuah variabel referensi obyek tidak selalu terpaku menunjuk / mereferensi suatu obyek tertentu. Variabel ini dapat saja mereferensi obyek lain. Seperti diperlihatkan pada baris ke-8 pada Contoh 6.4, pernyataan :

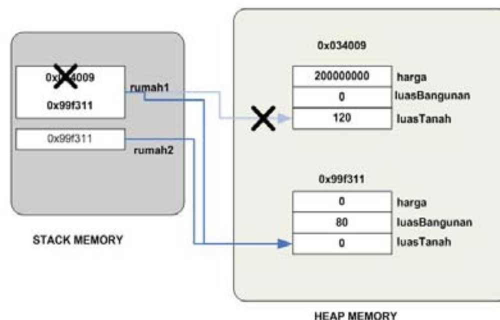
```
rumah1 = rumah2;
```

menyatakan bahwa isi variabel *rumah2* di-*assign* ke variabel *rumah1*. Dengan demikian variabel referensi *rumah1* tidak lagi menunjuk ke obyek semula, tetapi menunjuk obyek yang ditunjuk oleh variabel *rumah2*.

Contoh 6.4.

```
 1   //File : AplikasiMakelarRumah.java
 2   public class AplikasiMakelarRumah{
 3          public static void main(String [] args ){
 4                 Rumah rumah1;
 5                 Rumah rumah2;
 6                 rumah1 = new Rumah( );
 7                 rumah2 = new Rumah( );
 8                 rumah1 = rumah2;
 9                 rumah1.luasTanah = 120;
10                 rumah2.luasBangunan = 80;
11                 rumah1. harga = 200000000;
12          }
13   }
```

Hasil akhir program dapat dilihat pada Gambar 6.5.



Gambar 6.5. Pemindahan Referensi dari *rumah1* ke *rumah2*

EKSPERIMEN

Tambahkan baris program pada main method pada Contoh 6.4, setelah baris ke-11 sebagai berikut :

```
System.out.println(rumah1==rumah2);
```

Perhatikan output yang dihasilkan program ! Jika outputnya adalah *false,* variabel referensi rumah1 menunjuk obyek yang berbeda dengan obyek yang ditunjuk oleh variabel referensi rumah2. Jika outputnya adalah *true*, maka variabel referensi rumah1 menunjuk obyek yang sama dengan obyek yang ditunjuk oleh variabel referensi rumah2.

7 MENGGUNAKAN CLASS STRING DAN PUSTAKA JAVA

VII.1 Menggunakan Class *String*

String adalah class pada Java API yang paling unik diantara class-class lainnya dalam teknologi Java. Beberapa keunikan class *String* antara lain :

a. String merupakan kumpulan karakter-karakter. Variabel anggota dari class *String* adalah kumpulan variabel-variabel karakter yang berjumlah dari nol sampai berapapun (sampai memory tidak mencukupi).
b. String dapat diinstantiasi tanpa menggunakan kata kunci *new*, berbeda dengan obyek lain yang harus diinstantiasi dengan menggunakan kata kunci *new*.

Contoh pendeklarasian dan inisialisasi variabel *String* dapat dilihat pada Contoh 7.1.

<u>Contoh 7.1</u>

```
     //File : AplikasiSiswa.java

 1   public class AplikasiSiswa{
 2        public static void main (String[] args){
 3          String namaSiswa1 = "Adi";
 4          String namaSiswa2 = new String ("Adi");
 5          String namaSiswa3 = "Adi";
 6          String namaSiswa4 = new String("Adi");
 7
 8          System.out.println(namaSiswa1 == namaSiswa2);
 9          System.out.println(namaSiswa1 == namaSiswa3);
10          System.out.println(namaSiswa1.equals(namaSiswa2));
11          System.out.println(namaSiswa2 == namaSiswa4);
12          System.out.println(namaSiswa2.equals(namaSiswa4));
13        }
14   }
```

VII.1.1 Menginstanstiasi Obyek *String* dengan Kata Kunci *new*

Obyek *String* dapat diinstantiasi dengan menggunakan kata kunci *new*, seperti pada instantiasi obyek-obyek lainnya. Seperti pada baris ke-4 dan baris ke-6 pada Contoh 7.1 :

```
String namaSiswa2 = new String (“Adi”);
String namaSiswa4 = new String(“Adi”);
```

Ada perbedaan pada representasi memory antara instanstiasi obyek *String* menggunakan *new* dengan obyek lainnya. Pada instanstiasi obyek lain, akan dibentuk 1 obyek pada Heap Memory yang memuat variabel-variabel atribut / anggota.

Instanstiasi obyek String menggunakan *new* akan membentuk 2 buah obyek, yaitu :

a. obyek String, yang memuat referensi ke suatu *String literal* pada *literal pool*.
b. String literal, yang memuat karakter-karakter. String literal ini terletak pada *literal pool*.

Catatan : *Literal Pool* adalah satu blok alokasi memory pada Heap Memory yang khusus berisi kumpulan String literal. Alokasi *literal pool* ini dimaksudkan untuk mengakomodasi apabila terdapat lebih dari satu obyek String yang mereferensi ke literal yang sama, tidak perlu membuat 2 *string literal* dengan komposisi karakter yang sama, tetapi cukup hanya 1 *string literal* saja.

Pada baris ke-4, obyek *namaSiswa2* diinstanstiasi dengan representasi karakter = “Adi”. Pada proses instanstiasi ini, JVM akan membentuk obyek String dan *String Literal* pada *literal pool*. Isi *literal pool* ini adalah representasi karakter dari *namaSiswa2, yaitu* "Adi". Obyek String bernilai alamat *string literal* tersebut.

Kemudian pada baris ke-6, obyek *namaSiswa4* diinstanstiasi dengan representasi karakter = “Adi”. Pada proses instanstiasi ini, JVM akan membentuk obyek String dan *String Literal* pada *literal pool*. Isi *literal pool* ini adalah representasi karakter dari *namaSiswa4, yaitu* "Adi". Obyek String bernilai alamat *string literal* tersebut.

Meskipun representasi karakternya sama-sama “Adi”, tetapi *String literal* untuk *namaSiswa2* tidak sama dengan *String literal*, sehingga ketika baris ke-11 dieksekusi, maka akan menghasilkan nilai *false.*

Jika kita ingin membandingkan *namaSiswa2* dan *namaSiswa4* dengan logika : “jika nama kedua siswa itu sama, berarti kedua siswa tersebut sebenarnya adalah siswa yang sama “, maka kita harus menggunakan method *equals( )* seperti pada baris ke-12.

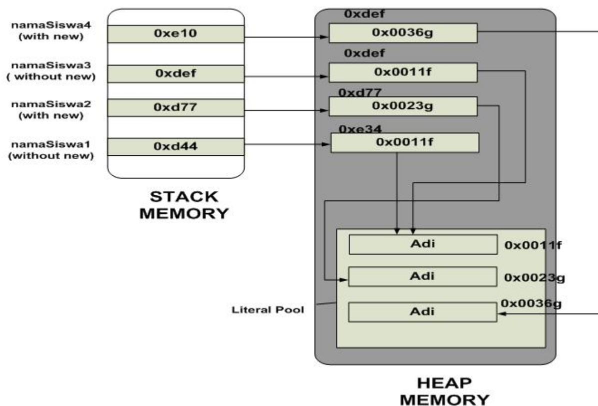
VII.1.2 Menginstantiasi Obyek *String* tanpa Kata Kunci *new*

Berbeda dengan instanstiasi obyek *String* dengan menggunakan *new*, instanstiasi obyek *String* tanpa menggunakan *new* hanya membentuk 1 obyek saja, yaitu *obyek String*. *String literal* dibentuk hanya apabila pada *literal pool* tidak ditemukan *String literal* yang representasi karakternya sama dengan representasi karakter obyek String.

Pada baris ke-3 Contoh 7.1, obyek *namaSiswa1* diinstanstiasi dengan representasi karakter = "Adi". Pada proses instanstiasi ini, JVM akan membentuk obyek String. Kemudian JVM akan memeriksa *literal pool* untuk melihat apakah sudah ada *string literal* dengan representasi karakter yang sama dengan *namaSiswa1*. Karena belum ada *string literal* dengan representasi karakter yang diinginkan, maka JVM akan membentuk *string literal* pada *literal pool* dengan representasi karakter = "Adi". Lalu obyek String akan diisi dengan alamat *string literal* tersebut.

Kemudian pada baris ke-5 Contoh 7.1, obyek *namaSiswa3* diinstanstiasi dengan representasi karakter = "Adi". Pada proses instanstiasi ini, JVM akan membentuk obyek String. Kemudian JVM akan memeriksa *literal pool* untuk melihat apakah sudah ada *string literal* dengan representasi karakter yang sama dengan *namaSiswa3*. Ternyata sudah ada *string literal* tersebut. Oleh karena itu JVM tidak membentuk *string literal* , dan mengisi nilai obyek String dengan alamat *string literal* tersebut (yang juga direferensi oleh obyek String *namaSiswa1*).

Hasil akhir dari program pada Contoh 7.1 dapat dilihat pada Gambar 7.1.



Gambar 7.1. Instanstiasi Obyek String

VII.1.3 Penggunaan Operator '==' dan Method *equals ( )* untuk Membandingkan Dua Buah String

Untuk membandingkan dua buah obyek String, dapat dilakukan dengan menggunakan operator '==' atau method *equals( )*.

Operator '==' lebih menekankan apakah kedua obyek String tersebut menunjuk ke *string literal* yang sama. Jika kita perhatikan Gambar 7.1, maka jika baris ke-8 Contoh 7.1 dieksekusi, maka akan menghasilkan nilai *false*, karena *string literal namaSiswa1* dan *namaSiswa2* tidak sama (meskipun representasi karakternya sama).

Sedangkan method *equals( )* lebih menekankan apakah representasi karakter kedua String sama atau tidak. Jika kita perhatikan Gambar 7.1, maka jika baris ke-10 dieksekusi, maka hasilnya akan menjadi *true*, meskipun *string literal namaSiswa1* dan *namaSiswa2* tidak sama.

VII.1.4 Menggunakan Variabel Referensi untuk Obyek *String*

Variabel Referensi untuk Obyek *String* dapat berlaku seperti variabel primitif, seperti pada Contoh 7.2.

Contoh 7.2

```
//File : Person.java

1   public class Person{
2        public static void main (String[] args){
3              String nama = “Surya”;
4              String kota = “Bandung”;
5              System.out.println(“Nama saya ”+ nama +
6              “, tinggal di “ + kota);
7        }
8   }
```

Contoh 7.2 akan menghasilkan output :

```
Nama saya Seno, tinggal di Bandung
```

EKSPERIMEN

Apakah ekspresi String berikut ini valid (benar, dapat dikompilasi) ?

a. String jumlah = new String (“jumlah = “ + 200) ;
b. String nama = “Rano ” + “Karno”;
c. String word = “Hello ”; word += “World” ;

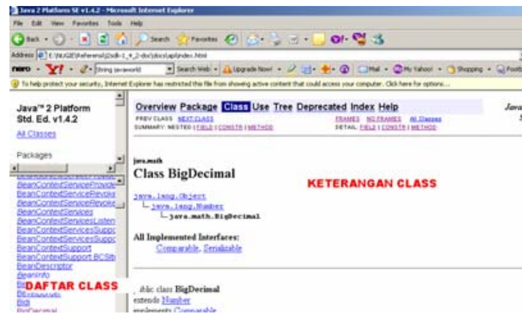
VII.2 Investigasi Pustaka Class Java

Java Teknologi selalu dikemas dalam bentuk Software Development Kit (SDK), yang berisi class-class yang telah disediakan dahulu oleh Sun Microsystems. Class-class ini membentuk suatu Application Programming Interface (API) Java , yang menjembatani programmer dengan SDK dalam mengembangkan aplikasi.

Untuk dapat memahami class-class pada API Java, diperlukan pustaka / dokumentasi yang menerangkan cara menggunakan class-class tersebut.

VII.2.1 Spesifikasi Pustaka Class-class Java

Spesifikasi Pustaka Class-class Java adalah dokumen yang menerangkan kegunaan class-class pada API Java. Spesifikasi ini dibuat berdasarkan versi SDK-nya. Misalnya, SDK1.3 mempunyai spesifikasi pustaka sendiri, SDK1.4 mempunyai pustaka sendiri, dan seterusnya. Gambar 7.2 memperlihatkan tampilan Spesifikasi Pustaka Class Java.



Gambar 7.2 Spesifikasi Pustaka Java

Pada pustaka Java terdapat 2 bagian utama, yaitu *daftar class* dan halaman *keterangan class*.

Pada Daftar Class, terdapat keterangan tentang class yang diseleksi, melingkupi keterangan tentang *field (* variabel publik statik), dan method.

VII.2.2 Menggunakan Spesifikasi Pustaka Class-class Java untuk Mempelajari Method

Spesifikasi Pustaka Class Java dapat digunakan untuk mempelajari method. Sebagai contoh, untuk mempelajari method yang sudah kita gunakan :

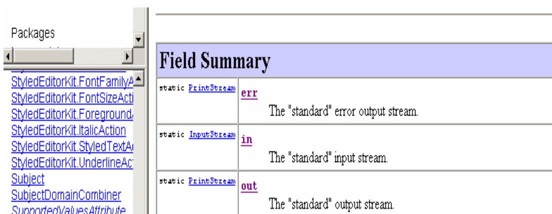
```
System.out.println ("Test");
```

Langkah-langkah yang harus dilakukan adalah :

a. Gunakan web browser untuk masuk ke halaman utama dari dokumentasi J2SE API.

b. Pada bagian *daftar class*, cari nama class *System*, lalu kliklah. Maka pada halaman *Keterangan class* akan muncul keterangan tentang class *System*.
   Jika sudah terbiasa dengan konvensi penamaan class, maka pemanggilan method yang didahului dengan identifier yang diawali dengan huruf besar, besar kemungkinan identifier itu adalah nama class. *System*, merupakan nama class. Oleh karena itu langkah pertama adalah mencari nama class *System*.

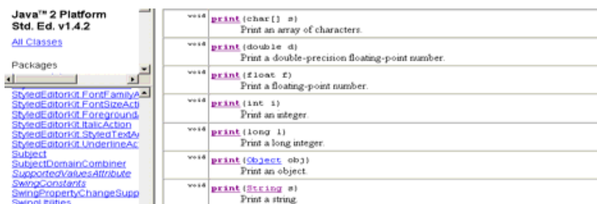
c. Pada bagian *Keterangan class*, cari pada bagian *Field*, nama variabel *out*. Lihat Gambar 7.3.



Gambar 7.3. Daftar Field pada class System

d. Perhatikan bahwa variabel *out* merupakan variabel statik yang berasal dari class *PrintStream*. Oleh karena itu, kliklah *PrintStream*. Maka akan muncul halaman keterangan class PrintStream.

e. Pada halaman keterangan class PrintStream, carilah method *println ()* pada daftar method. Lihat Gambar 7.4.



Gambar 7.4 Method *println* pada class PrintStream

f. Pada daftar method, terdapat banyak method *println( )*. Method *println* yang kita cari adalah method *println( )* yang memiliki argumen String, yaitu println (String s).

8 MENGGUNAKAN OPERATOR DAN KONSTRUKSI PENGAMBILAN KEPUTUSAN

VIII.1 Menggunakan Operator Relasional dan Kondisional

Dalam banyak kasus pemrograman Java, muncul kebutuhan-kebutuhan untuk mengevaluasi suatu variabel. Evaluasi variabel dibutuhkan untuk memutuskan arah aliran program. Untuk melakukan evaluasi terhadap satu / lebih variabel, digunakan operator relasional yang akan membandingkan variabel-variabel yang akan dievaluasi dengan nilai-nilai tertentu.

Sedangkan operator kondisional merupakan operator-operator yang mengatur arah aliran program berdasarkan hasil evaluasi terhadap satu / beberapa variabel.

VIII.1.1 Operator Relasional

Operator relasional, seperti yang diperlihatkan pada Tabel 8.1, berfungsi membuat suatu evaluasi terhadap variabel. Hasil evaluasinya akan menghasilkan jawaban *true*, jika pernyataan yang dibentuk oleh operator relasional adalah benar, dan *false*, jika pernyataan yang dibentuk oleh operator relasional adalah salah.

Misalnya, pada baris ke-2 kolom *Example* di Tabel 8.1, terdapat nilai i =1. Nilai i ini dibandingkan dengan 1 menggunakan operator relasional '!=', yang akan menghasilkan evaluasi :

```
i != 1
```

Hasil evaluasi ini jika dicetak ke layar, akan menghasilkan nilai *false*, karena pernyataan bahwa ' nilai 1 tidak sama dengan nilai 1' adalah pernyataan yang salah secara matematis.

Sedangkan pada baris ke-4 kolom *Example* pada Tabel 8.1, terdapat nilai i=1. Nilai i ini dibandingkan dengan 1 menggunakan operator relasional '<=' yang menghasilkan evaluasi :

```
i<=1
```

Hasil evaluasi ini jika dicetak ke layar, akan menghasilkan nilai *true*, karena pernyataan bahwa 'nilai 1 lebih kecil atau sama dengan 1' adalah pernyataan yang benar secara matematis.

Tabel 8.1
Operator Relasional

Condition	Operator	Example
Is equal to (atau "is the same as")	==	int i = 1; System.out.println(i==1); *// (output : true)*
Is not equal to (atau "is not the same as")	!=	int i = 1; System.out.println(i!=1); *// (output : false)*
Is less than	<	int i = 1; System.out.println(i<1); *// (output : false)*
Is less than or equal to	<=	int i = 1; System.out.println(i<=1); *//(output : true)*
Is greater than	>	int i = 1; System.out.println(i>1); *//(output : false)*
Is greater than or equal to	>=	int i = 1; System.out.println(i>=1); *// (output : true)*

VIII.1.2 Operator Kondisional

Operator kondisional merupakan operator yang membandingkan suatu kondisi dengan kondisi lain. Sama dengan operator relasional, operator kondisional akan menghasilkan nilai *boolean*, yaitu *true* atau *false*. Perbedaannya adalah, operator kondisional memiliki *operand* yang juga bernilai *boolean*, sedangkan operator relasional memiliki *operand* yang bernilai **bukan-*boolean*.**

Tabel 8.2
Operator Kondisional

Condition	Operator	Example

If one condition AND another condition	&&	int i = 1;
		int j = 2;
		System.out.println(
		(i<1)&&(j>0)
		);
		// (output : false)

Condition	Operator	Example

If either condition OR another condition	||	int i = 1;
		int j = 2;
		System.out.println(
		(i<1)||(j>0)
		);
		// (output : true)

NOT	!	int i = 1; System.out.println(!(i<3)); *// (output : true)*

VIII.2 Konstruksi Pengambilan Keputusan

Konstruksi pengambilan keputusan adalah konstruksi yang memungkinkan program melakukan evaluasi terhadap variabel / kondisi kemudian menjalankan alur program yang sesuai dengan kondisi. Dalam hal ini, program dikatakan *mengambil keputusan* berdasarkan hasil evaluasi variabel /kondisi.

Ada beberapa konstruksi pengambilan keputusan, yaitu :

a. Konstruksi *if*
b. Konstruksi *if..else*
c. Konstruksi *switch*

VIII.2.1 Konstruksi *if*

Konstruksi *if* merupakan bentuk konstruksi pengambilan keputusan dengan 2 kemungkinan keputusan. Kemungkinan-kemungkinan keputusan itu akan dipilih berdasarkan suatu kondisi yang diperiksa. Kondisi tersebut merupakan suatu ekspresi boolean / *boolean expression*.

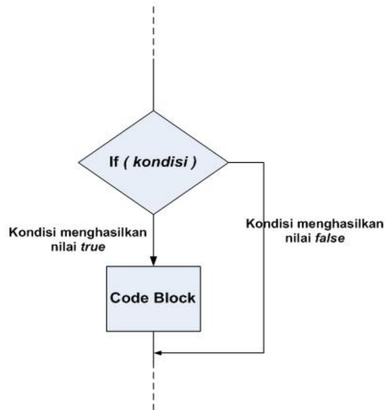
Syntax dasar dari konstruksi *if* adalah sebagai berikut :

```
if ( boolean_expression ){
        code block;

} //akhir dari konstruksi 'if'
 //program dilanjutkan
```

di mana :

a. *boolean_expression* adalah kombinasi dari operasi-operasi relasional, kondisional. Nilai yang dihasilkan oleh *boolean_expression* adalah *true* atau *false*.
b. *code_block* merepresentasikan baris-baris program yang akan dieksekusi apabila nilai *boolean_expression* adalah *true*.



Gambar 8.1 Diagram Alir untuk Konstruksi *if*

Jika *boolean_expression* bernilai benar, maka baris-baris program yang berada dalam *code_block* akan dieksekusi. Setelah *code_block* dieksekusi, maka baris-baris program yang akan dijalankan adalah program di luar konstruksi *if.*

Jika *boolean_expression* bernilai salah, maka baris-baris program yang berada dalam *code_block* tidak dieksekusi, dan program berikutnya yang dijalankan adalah program di luar konstruksi *if.*

Proses pengambilan keputusan dengan menggunakan konstruksi *if* dapat divisualisasikan menggunakan diagram alir, seperti pada Gambar 8.1.

Contoh pengambilan keputusan dapat dilihat pada Contoh 8.1.

Contoh 8.1

```
 1  //File : HasilUjian.java
 2  public class HasilUjian{
 3        public static void main(String[] args){
 4              int nilai1 = 8;
 5              int nilai2 = 7;
 6              int nilai3 = 5;
 7              float rata_rata = (float)(nilai1 + nilai2
 8                    + nilai3)/3;
 9              if(rata_rata<5){
10                    System.out.println("Tidak Lulus");
11              }
12              System.out.println("Nilai    Rata-rata    =
13              "+rata_rata);
14        }
15  }
```

Contoh 8.1 adalah program pengambilan keputusan apakah seorang siswa lulus atau tidak lulus. Keputusan tersebut diambil

berdasarkan nilai rata-rata yang diambil dari hasil-hasil ujian : *nilai1, nilai2,* dan *nilai3*.Rumus nilai rata-rata dinyatakan pada baris ke-7.

Proses evaluasi nilai rata-rata menggunakan *if* diperlihatkan pada baris ke-8, dengan *boolean_expression*-nya adalah :

```
rata-rata < 5
```

Pada Contoh 8.1 nilai rata-rata adalah 6,6666665, sehingga *boolean_expression* ' rata-rata < 5 ' akan bernilai *false*, sehingga, *code_block* pada baris ke-10 tidak dieksekusi. Eksekusi program berikutnya adalah eksekusi baris ke-12.

Hasil keluaran dari program Contoh 8.1 adalah :

```
Nilai Rata-rata = 6.6666665
```

EKSPERIMEN

2. Gantilah nilai pada nilai1 menjadi 2. Catatlah keluaran / output yang dihasilkan.
3. Ubahlah syarat ketidaklulusan : seseorang akan dinyatakan tidak lulus jika nilai rata-ratanya < 5 DAN salah satu di antara nilai1, nilai2,nilai3 ada yang bernilai <5.

VIII.2.2 Konstruksi *if / else*

Konstruksi *if / else* digunakan sebagai konstruksi pengambilan keputusan yang memiliki beberapa kemungkinan keputusan.

Syntax dasarnya adalah seperti berikut :

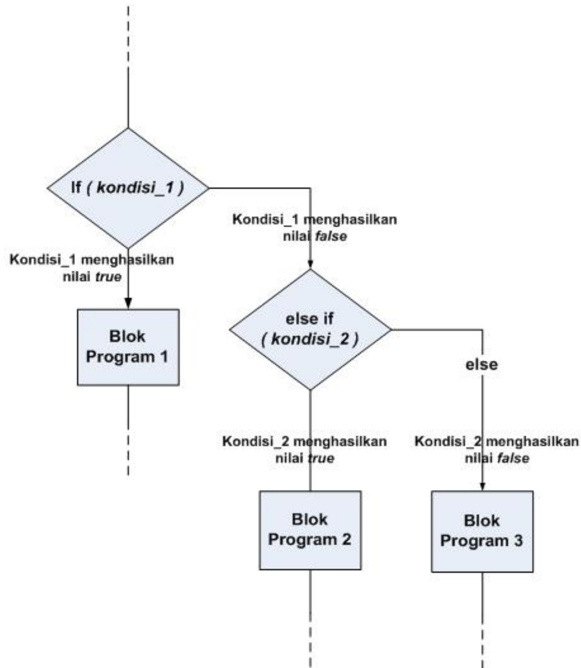
```
if ( boolean_expression_1 ){
        code block 1;

}
else if ( boolean_expression_2 ){
        code block 2;
}
            .
            .
            .
else{
        code block n;

}
```

di mana :

a. *boolean_expression* adalah kombinasi dari operasi-operasi relasional, kondisional. Nilai yang dihasilkan oleh *boolean_expression* adalah *true* atau *false.*
b. *code_block k* merepresentasikan baris-baris program yang akan dieksekusi apabila nilai *boolean_expression_k* adalah *true*.
   k = 1,2,...,n



Gambar 8.2 Diagram Alir untuk Konstruksi if-else

Contoh 8.2

```
 1   //File : HasilUjian.java
 2   public class HasilUjian{
 3          public static void main(String[] args){
 4                 int nilai1 = 8;
 5                 int nilai2 = 7;
 6                 int nilai3 = 5;
 7                 float rata_rata =
 8                       (float)(nilai1 + nilai2 + nilai3)/3;
 9                 if(rata_rata<5){
10                       System.out.println("Tidak Lulus");
11                 }
12                 else if((rata_rata>=5) && (rata_rata<6)){
13                       System.out.println("Harus ikut "+
14                       "ujian perbaikan");
15                 }
16                 else{ // rata_rata >= 6
17                       System.out.println("Lulus");
18                 }
19                 System.out.println("Nilai Rata-rata =
20                                     "+rata_rata);
21          }
     }
```

Pada Contoh 8.2, terdapat 3 kondisi yang akan dievaluasi, yaitu :

a. rata-rata < 5
b. rata-rata $\geq$ 5 DAN rata-rata < 6
c. kondisi di luar a. dan b. , yaitu rata-rata $\geq$ 6

Pada baris ke-7, nilai rata-rata yang disimpan pada variabel *rata-rata* adalah 6,6666665. Nilai variabel *rata_rata* ini akan dievaluasi pada baris-baris berikutnya untuk menentukan pesan apa yang akan ditampilkan di layar monitor.

Pertama-tama variabel *rata_rata* dievaluasi pada baris ke-9, dengan *boolean_expression* sebagai berikut :

```
rata_rata < 5
```

Hasil evaluasinya akan bernilai *false,* karena jelas bahwa seharusnya 6,6666665 lebih besar daripada 5, dan bukan lebih kecil. Karena hasil evaluasi bernilai *false,* maka *code block* pada baris ke-10 tidak dieksekusi.

Kemudian dilakukan evaluasi untuk kondisi kedua, yang dinyatakan dengan *boolean_expression* pada baris ke 12 :

```
( rata_rata>=5) && (rata_rata <6)
```

Pada evaluasi kedua, terdapat *boolean_expression* yang merupakan ekspresi kondisional, yang membandingkan dua kondisi dalam operator && (AND / dan), di mana nilai dari ekspresi akan benar jika dan hanya jika kedua kondisi yang dibandingkan juga bernilai benar.

Kondisi pertama : *rata_rata>=5*, akan menghasilkan nilai *true*, karena 6,6666665 memang lebih besar daripada 5.

Pada kondisi kedua : *rata_rata < 6*, akan menghasilkan nilai *false*, karena 6,6666665 seharusnya lebih besar daripada 6, bukan lebih kecil daripada 6.

Karena kondisi pertama bernilai *true* dan kondisi kedua bernilai *false*, maka *boolean_expression* menjadi :

```
true && false
(true AND false)
```

Dengan demikian, hasil *boolean_expression* adalah *false*. Berarti hasil evaluasi juga bernilai *false*. Karena hasil evaluasi bernilai *false,* maka *code block* pada baris ke-13 sampai 14 tidak dieksekusi.

Selanjutnya evaluasi akan berlanjut ke blok *else* berikutnya, pada baris ke-16. Pada baris ini tidak disebutkan kondisi yang akan diuji :

```
else{
```

Hal ini berarti, kondisi yang diuji adalah kondisi di luar semua kondisi yang telah diuji. Kondisi yang telah diuji adalah *rata_rata < 5, (rata_rata>=5)&&(rata_rata<6)*. Kondisi di luar itu tentu saja adalah :

```
rata_rata >=6
```

Karena nilai *rata_rata* adalah 6,6666665, maka pernyataan *rata_rata >= 6* akan menghasilkan nilai *true*.

Karena hasil evaluasi menghasilkan nilai *true*, maka *code block* pada baris ke-17 akan dieksekusi :

```
System.out.println("Lulus");
```

Setelah baris ke-17 dieksekusi, maka eksekusi program berikutnya adalah baris ke-19 :

```
System.out.println("Nilai rata-rata = " + rata_rata);
```

Program pada Contoh 8.2 akan menghasilkan output berikut :

```
Lulus Nilai Rata-rata = 6.6666665
```

EKSPERIMEN

3. Gantilah nilai pada 'nilai2' pada Contoh 8.2 menjadi 0. Catatlah keluaran / output yang dihasilkan.
4. Ubahlah syarat ketidaklulusan :
    a. Tidak lulus, bila nilai rata-rata < 5,5
    b. Harus ikut ujian ulang, bila nilai rata-rata ≥ 5,5 dan nilai rata-rata < 7
    c. Lulus, bila nilai rata-rata ≥ 7

    Catatlah output yang dihasilkan.

VIII.2.3 Konstruksi *Switch*

Konstruksi *switch* adalah konstruksi pengambilan keputusan yang mengevaluasi kemungkinan-kemungkinan nilai dari variabel yang dievaluasi.

Bentuk umum syntax konstruksi *switch* adalah sebagai berikut:

```
switch( variabel ) {
        case  nilai_literal_1 :
                code_block_1;
                [ break; ]
        case nilai_literal_2 :
                code_block_2;
                [ break; ]
        .
        .
        .
        [default:]
                code_block_default;
                [ break; ]
}
```

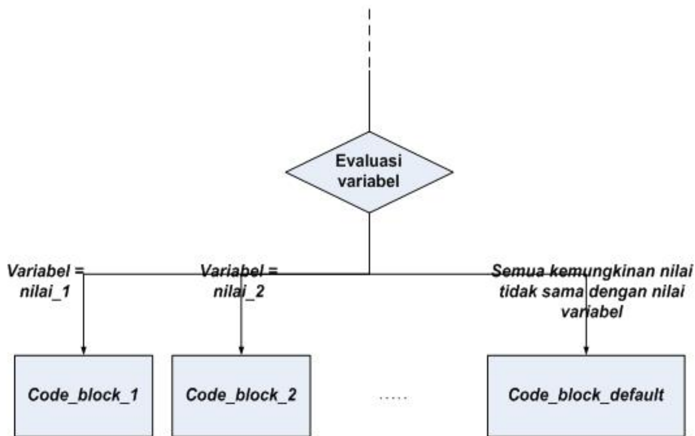
di mana :

a. *switch* adalah kata kunci yang mengindikasikan dimulainya konstruksi *switch*.
b. *variabel* adalah variabel yang nilainya akan dievaluasi. *variabel* hanya dapat bertipe-data *char, byte, short,* atau *int.*
c. *case* adalah kata kunci yang mengindikasikan sebuah nilai yang diuji. Kombinasi kata kunci *case* dan *nilai_literal* disebut *case label*.
d. *nilai_literal_k* adalah nilai yang mungkin akan menjadi nilai variabel. *nilai_literal_k* tidak dapat berupa variabel, ekspresi, atau method, tetapi dapat merupakan konstanta.
   k = {default, 1,2,...,n }

e. [*break;*] adalah pernyataan yang sifatnya opsional, yang mengakibatkan aliran program keluar dari blok *switch*. Jika setelah *code_block_k* tidak terdapat pernyataan *break;* , maka aliran program akan masuk ke case berikutnya.
   k = {default, 1,2,3,...,n}

f. *default* adalah kata kunci yang mengindikasikan *code_block-_default* akan dieksekusi jika semua case yang diuji tidak sesuai dengan nilai *variabel* .

Diagram alir dari konstruksi *switch* ditunjukkan pada Gambar 8.3.



Gambar 8.3 Diagram Alir untuk Konstruksi *switch*

Contoh 8.3

```
 1  //File : HasilUjian.java
 2  public class HasilUjian{
 3          public static void main(String[] args){
 4                  int nilai1 = 8;
 5                  int nilai2 = 7;
 6                  int nilai3 = 5;
 7                  float rata_rata =
 8                          (float)(nilai1 + nilai2 + nilai3)/3;
 9
10                  switch((int)rata_rata){
11                          case 0 :
12                          case 1 :
13                          case 2 :
14                          case 3 :
```

```
15                      case 4 :
16                         System.out.println("Tidak Lulus");
17                         break;
18                      case 5 :
19                         System.out.println("Harus ikut "+
20                                        "ujian perbaikan");
21                         break;
22                      default :
23                         System.out.println("Lulus");
24                         break;
25                }
26
27                System.out.println("Nilai Rata-rata =
28                                        "+rata_rata);
29          }
    }
```

Pada Contoh 8.3, setelah nilai variabel *rata_rata* dikalkulasi pada baris ke-7, dilakukan evaluasi terhadap *rata_rata* baris ke-10 :

```
switch((int)rata_rata){
```

Pada baris 10, variabel *rata_rata* dievaluasi nilainya, dan dibandingkan dengan kemungkinan-kemungkinan nilai. Kemungkinan-kemungkinan nilai tersebut adalah 1, 2, 3, 4, 5 dan selebihnya (6,7,8, dst).

Nilai variabel *rata_rata* pada baris ke-7 adalah 6,6666665. Nilai variabel *rata_rata* ini sebelum diuji pada baris ke-10 di-*casting* ke tipe data *int*, sehingga nilanya menjadi 6 (sisa desimalnya dihilangkan).

Pengujian dilakukan dengan pertama-tama membandingkan nilai *rata_rata* ter-*casting* dengan nilai pada *case* yang pertama, yaitu nilai literal 0. Karena nilai literal pada *case* pertama tidak sama dengan nilai *rata_rata* ter-*casting*, maka evaluasi dilanjutkan ke *case* berikutnya. Hal yang sama terjadi juga pada *case* kedua. Karena pada *case* pertama (0) sampai *case* keenam (5) tidak didapatkan hasil *true*, maka pemeriksaan dilanjutkan ke nilai di luar { 0,1,...,5 }, yaitu *default*.

Pada pengujian *default*, evaluasi menghasilkan nilai *true*, karena nilai *rata_rata* ter-*casting* berada di luar { 0, 1, 2,..., 5 }. Karena menghasilkan nilai *true,* maka *code block* yang dijalankan adalah yang dilingkupi oleh blok *default*. Proses akan keluar dari block *switch* setelah mengeksekusi pernyataan *break* pada baris 24.

EKSPERIMEN

5. Gantilah nilai pada 'nilai 1' dan 'nilai 2' pada Contoh 8.3 menjadi 0.
   Kemudian modifikasilah baris ke-12 sebagai berikut :

   ```
   12   case 1  :  System.out.println("Nilai rata-rata = 1");
   ```

   Catatlah keluaran / output yang dihasilkan.

6. Komposisi nilai seperti pada eksperimen 1, tetapi modifikasi baris ke-12 adalah sebagai berikut :

   ```
   12    case 1  : System.out.println ("Nilai rata-
                     rata = 1"); break;
   ```

   Catatlah keluaran / output yang dihasilkan.
   Bandingkanlah dengan hasil pada eksperimen 1.
   Mengapa demikian ?

9

MENGGUNAKAN KONSTRUKSI LOOP

IX.1 Konstruksi Loop

Konstruksi Loop adalah konstruksi yang digunakan untuk mengakomodasi pengulangan proses. Konstruksi Loop diperlukan untuk lebih mengefisienkan penulisan kode program, sehingga tidak perlu dilakukan pengulangan penulisan kode untuk merepresentasikan suatu proses yang berulang.

Ada beberapa konstruksi Loop yang digunakan dalam bahasa pemrograman Java, antara lain :

a. Konstruksi *while*
b. Konstruksi *for*
c. Konstruksi *do/while*

IX.2 Membuat Loop Menggunakan *While*

Konstruksi *while* adalah konstruksi loop yang jumlah perulangannya tergantung pada suatu kondisi logika tertentu. Bentuk umum konstruksi *while* adalah sebagai berikut :

```
while(boolean_expression){

        code_block;

} //akhir dari konstruksi while

  //program dilanjutkan di sini
```

Pada konstruksi *while*, langkah-langkah proses perulangannya adalah sebagai berikut :

a. Sistem memeriksa *boolean_expression*.
b. Jika nilai *boolean_expression* adalah *true*, maka *code_block* akan dieksekusi. Jika tidak, maka *code_block* tidak dieksekusi.
c. Jika di dalam *code_block* terdapat pernyataan kondisi yang menyebabkan proses harus keluar dari blok *while*, maka proses akan keluar dari loop, meskipun *boolean_expression* masih bernilai *true*.

Pada Contoh 9.1 diperlihatkan contoh penggunaan konstruksi *while*. Pada baris ke-3 sampai ke-5, dideklarasikan 3 buah variabel *num1, num2,* dan *num3*. Variabel *num1* diberi nilai 0, *num2* diberi nilai 23, dan *num3* diberi nilai yang merupakan hasil penjumlahan *num1* dan *num2* yang menghasilkan nilai 23.

Kemudian pada baris ke-6 sampai dengan ke-11 terdapat konstruksi *while*, dengan pemeriksaan kondisi dilakukan baris ke-6 :

```
while ( num3 > num 1 )
```

Pada pemeriksaan pertama kali, *num3* memiliki nilai 23, dan *num1* memiliki nilai 0, sehingga *boolean expression " num3 > num1 "* bernilai *true*. Oleh karena itu *code block* pada baris ke-7 sampai ke-10. Pada baris ke-7, nilai *num2* diperbaharui dengan mengurangkan nilai *num2* semula dengan 3. Pada baris ke-8, nilai *num1* diperbaharui dengan menambahkan nilai *num1* tersebut dengan 2. Pada baris ke-9, nilai *num3* diperbaharui dengan mengambil hasil penjumlahan *num1* dan *num2*.

Proses pembaharuan nilai ketiga variabel *num1, num2,* dan *num3* dapat dilihat pada Tabel 9.1

Tabel 9.1
Proses Pembaharuan *num1, num2,* dan *num3*

Loop ke-	num1	num2	num3	num3 > num1
0 (sebelum loop dimulai)	0	23	23	true (loop dimulai dari sini)
1	2	20	22	true
2	4	17	21	true
3	6	14	20	true
4	8	11	19	true
5	10	8	18	true
6	12	5	17	true
7	15	2	17	true
8	18	-1	17	false
9	keluar dari loop			

Contoh 9.1

```
1   public class Contoh9_1 {
2      public static void main(String[] args){
3           int num1 = 0;
4           int num2 = 23;
5           int num3 = num1+num2;
6           while(num3 > num1){
7                 num2-=3;
8                 num1+=2;
9                 num3 = num1+num2;
10          System.out.println("num1="+num1 +",num3="+num3);
11          }
12     }
13  }
```

EKSPERIMEN

1. Buatlah *boolean expression* pada baris ke-6. Lalu catatlah hasilnya.
2. Buatlah sebuah aplikasi sederhana yang menggunakan skema *while*. Misalnya :
   a. Aplikasi yang menghitung besarnya bunga tabungan selama n-tahun dengan besarnya bunga tahunan sebesar p %.
   b. Aplikasi yang menampilkan simbol '#' yang dicetak berkali-kali sebanyak n-kali.

Pada Contoh 9.2 diperlihatkan *boolean_expression* yang langsung diberi nilai *true*, sehingga sampai kapanpun *boolean_expression* ini tidak akan pernah bernilai *false*. Akibatnya, looping akan terus dieksekusi sampai jumlah loop yang tidak terbatas.

Agar looping menjadi berhingga, maka harus ada tambahan pemeriksaan kondisi di dalam blok *while* yang akan mengakibatkan looping berakhir. Solusi ini diperlihatkan pada baris ke-7 sampai ke-9. Pada baris ke-7 diperlihatkan pemeriksaan nilai *variable*, apakah nilainya lebih kecil daripada 10, dengan pernyataan :

```
if (variable<10)
```

Jika pemeriksaan ini menghasilkan nilai *true*, maka baris ke-8 akan dieksekusi.

Contoh 9.2

```
 1  public class Contoh9_2 {
 2     public static void main(String[] args){
 3         int variable = 20;
 4         while(true){
 5           System.out.println("Nilai variable = "+variable);
 6           --variable;
 7           if(variable<10){
 8              break;
 9           }
10         }
11     }
12  }
```

EKSPERIMEN

3. Hilangkanlah baris ke-7 sampai ke-9. Catat apa yang terjadi. Mengapa demikian ?

IX.3 Membuat Loop Menggunakan *For*

Konstruksi Loop menggunakan *for* adalah bentuk lain konstruksi loop selain *while*. Perbedaannya adalah, pada *for,* terdapat 3 segmen yang dipertimbangkan, yaitu :

a. segmen inisialisasi, yang berisi pernyataan pemberian nilai awal untuk suatu variabel parameter.
b. segmen *boolean_expression,* yang berisi pernyataan logika yang akan diperiksa, sebagai syarat looping terus dilanjutkan. Looping akan dilanjutkan jika nilai ekspresi boolean pada segmen ini bernilai *true*.
c. segmen update, yang berisi pernyataan updating parameter ketika satu putaran pada loop selesai dieksekusi.

Bentuk umum dari konstruksi loop menggunakan *for* adalah :

```
for(initialize[,initialize] ; boolean_expression ;
                update[,update]){
      code_block;
 }
```

Dari bentuk umum di atas dapat disimpulkan bahwa :

a. segment inisialisasi dapat diisi dengan lebih dari 1 pernyataan inisialisasi

b. segmen *boolean_expression* hanya dapat diisi oleh 1 pernyataan logika
c. segmen *update* dapat diisi dengan lebih dari 1 pernyataan update

Contoh 9.3 menunjukkan penggunaan konstruksi loop *for*. Pada baris ke-3 diperlihatkan bahwa :

a. segmen inisialisasi diisi oleh pernyataan "int variable=20". Ini berarti sebelum looping dilaksanakan, sebuah variabel *variabel* dideklarasikan dan diinisialisasi dengan nilai 20.
b. segmen *boolean_expression* diisi dengan pernyataan "variable>=10"
c. segmen update diisi dengan pernyataan "variable--" , yang berarti, untuk 1 kali putaran yang selesai dieksekusi, nilai *variable* akan dikurangi 1.

Contoh 9.3

```
1   public class Contoh9_3 {
2      public static void main(String[] args){
3        for(int variable=20; variable>=10;variable--){
4          System.out.println("Nilai variable = "+ variable);
5        }
6      }
7   }
```

Tabel 9.2 memperlihatkan proses pembaharuan nilai *variable*.

Tabel 9.2
Pembaharuan Nilai Variabel

Loop ke-	*variable*	*variable* >= 10
0	20	true
1	19	true
2	18	true
3	17	true
4	16	true
5	15	true
6	14	true
7	13	true
8	12	true
9	11	true
10	10	true
11	9	false
keluar dari loop		

EKSPERIMEN

4. Buatlah suatu aplikasi sederhana yang akan mencetak himpunan bilangan ganjil sampai 10 elemen.

Contoh 9.4 memperlihatkan segmen inisialisasi yang diisi dengan inisialisasi lebih dari 1 variabel, yaitu *variable1 = 20* dan *variable2=0*. Selain itu juga segmen update diisi dengan statemen update dari kedua variabel tersebut (baris ke-3 dan ke-4). Kondisi yang dinyatakan dalam segmen *boolean_expression* adalah variable1>=10 && variable2<=5.

Contoh 9.4

```
1   public class Contoh9_4{
2      public static void main(String[] args){
3         for(int variable1=20, variable2=0; variable1>=10&&
4               variable2<=5;variable1--,variable2++){
5            System.out.println("Nilai variable1= "+ variable1);
6            System.out.println("Nilai variable2= "+variable2);
7         }
8      }
9   }
```

Tabel 9.3
Progress Looping pada Contoh 9.4

Loop ke-	*variable1*	*variable2*	*variable1 >=10*	*variable2 <=5*	*variable1>=10 && variable2<=5*
1	20	0	true	true	true
2	19	1	true	true	true
3	18	2	true	true	true
4	17	3	true	true	true
5	16	4	true	true	true
6	15	5	true	true	true
7	14	6	true	false	false (keluar dari loop)

Pada Contoh 9.5 diperlihatkan penggunaan konstruksi loop *for* dengan semua segmen-nya tidak terisi. Bentuk ini sama dengan *while (true)*, yang akan mengakibatkan terjadinya looping tak berhingga.

Sama halnya dengan kasus *while* pada Contoh 9.2, bentuk *for (; ;)* mensyaratkan adanya tambahan pemeriksaan kondisi pada *code_block*-nya, supaya looping memiliki jumlah loop yang berhingga. Seperti yang terjadi pada baris ke-7. Looping akan berhenti ketika nilai *variable* < 10.

Contoh 9.5

```
 1  public class Contoh9_5{
 2     public static void main(String[] args){
 3     int variable = 20;
 4       for(  ;   ;  ){
 5         System.out.println("Nilai variable1= "+variable);
 6         variable--;
 7         if(variable<10)break;
 8       }
 9     }
10  }
```

IX.4 Membuat Loop Menggunakan *Do/While*

Konstruksi Loop *do/while* mirip dengan konstruksi *while*. Perbedaannya adalah pada urutan prosesnya, yaitu :

a. Looping dijalankan terlebih dahulu
b. Dilakukan pemeriksaan kondisi

Pada Contoh 9.6 diperlihatkan penggunaan konstruksi *do/while*. Pada baris ke-3, dilakukan deklarasi dan inisialisasi variabel *variable* dengan nilai 20.

Kemudian loop akan dilakukan satu kali, lalu dilakukan pemeriksaan *boolean_expression*. Karena *boolean_expression* pada baris ke-7 bernilai *false*, maka loop tidak dieksekusi lagi.

Pada konstruksi *while,* jika *boolean_expression* diset : "variable > 20", maka looping tidak akan dilakukan. Berbeda halnya dengan konstruksi loop *do/while*, dengan *boolean_expression* yang sama, looping akan dilakukan satu kali, sebelum akhirnya proses keluar dari loop.

Contoh 9.6

```
1   public class Contoh9_6{
2      public static void main(String[] args){
3      int variable = 20;
4       do{
5        System.out.println("Nilai variable1= "+ variable);
6        variable--;
7       }while(variable>20);
8      }
9   }
```

Hasil eksekusi pada Contoh 9.6 adalah :

```
Nilai variable = 20
```

IX.5 Loop Bersarang (*Nested Loop*)

Loop bersarang / *nested loop* adalah susunan looping bertingkat, di mana terdapat minimal satu blok loop di dalam blok loop yang lain. Contoh 9.7 memperlihatkan adanya blok loop *while* di dalam blok loop *for.*

Contoh 9.7

```
 1  public class Contoh9_7{
 2     public static void main(String[] args){
 3        int j=0;
 4        for(int i=0;i<5;i++){
 5             while(j<5){
 6                System.out.print(" @ ");
 7                j++;
 8             }
 9             j=0;
10             System.out.print("\n");
11        }
12     }
13  }
```

Contoh 9.7 akan menghasilkan sebuah bujursangkar dengan panjang 5 karakter dan lebar 5 karakter, sebagai berikut :

```
@@@@@
@@@@@
@@@@@
@@@@@
@@@@@
```

IX.6 Perbandingan Konstruksi Loop

Setiap konstruksi loop yang dikenal dalam pemrograman Java memiliki kegunaan masing-masing, antara lain :

a. *While*, digunakan untuk membuat iterasi dengan jumlah iterasi yang tidak pasti dan untuk iterasi dari *nol* sampai beberapa kali.
b. *Do/While,* digunakan untuk membuat iterasi dengan jumlah iterasi yang tidak pasti, dan untuk iterasi dari *satu* sampai beberapa kali.
c. *For*, lebih tepat digunakan untuk membuat iterasi dengan jumlah iterasi yang pasti dan berhingga.

IX.7 Pernyataan Continue

Programmer terkadang ingin meneruskan perulangan, tetapi menghentikan sisa proses pada program untuk iterasi yang bersangkutan. Hal ini dapat dilakukan dengan pernyataan `goto` yang memintas program, tetapi masih di dalam perulangan. Pernyataan `continue` di Java melakukan hal demikian. Pada pengulangan `while` dan `do-while`, pengaturan dipindahkan dari pernyataan `continue`, memintas sisa program untuk memeriksa pernyataan terminasi. Pada perulangan `for`, bagian ketiga pernyataan `for` akan segera dieksekusi setelah `continue`. Berikut ini contoh penggunaan `continue`.

Contoh 9.8

```
1  public class Continue{
2     public static void main(String[] args){
3        for(int i=0;i<10;i++){
4             System.out.print(i + " ");
5             if (i%2== 0) continue;
6             System.out.println(" ");
7        }
8     }
9  }
```

Pada contoh 9.8 menggunakan System.out.print untuk mencetak tanpa pindah baris. Kemudian digunakan operator mod, %, untuk memeriksa apakah indeksnya genap atau ganjil. Jika genap, pengulangan dilanjutkan tanpa mencetak perpindahan baris.

Seperti pernyataan `break`, `continue` dapat dilengkapi denga label untuk menentukan pengulangan mana yang harus dilanjutkan. Pernyataan `continue` pada contoh di bawah ini menghentikan perulangan `j` dan meneruskan iterasi selanjutnya pada perulangan i.

Contoh 9.9

```
 1  public class ContinueLabel{
 2     public static void main(String[] args){
 3     outer :    for(int i=0;i<10;i++){
 4                  for (int j=0;j<10;j++){
 5                      if (j>i){
 6                        System.out.println(" ");
 7                        continue outer;
 8                      }
 9                  System.out.print(" " + (i*j));
10                }
11              }
12              System.out.println("");
13      }
14  }
```

10 MENGGUNAKAN METHOD

X.1 Pengertian *Method*

Method adalah satu kontainer pada class yang memuat baris-baris kode. Semua baris kode pada pemrograman Java harus berada pada blok method, dan semua method harus berada di dalam blok class. Method biasanya digunakan untuk mengenkapsulasi proses-proses yang diperlukan untuk membentu suatu fungsi / tugas tertentu.

Contoh method dapat dianalogikan dengan sebuah fungsi matematika :

```
F ( x , y ) = x + y
```

Fungsi matematika ini, jika digunakan dalam operasi matematika :

```
b = F ( 2 , 4 )
```

akan menghasilkan nilai 6, karena pada definisi F (x , y), dinyatakan bahwa kedua parameter yang dilewatkan ke dalam fungsi, yaitu x dan y, akan dijumlahkan. Oleh karena itu, nilai 2 dan 4 yang dilewatkan ke fungsi F akan dijumlahkan, sehingga menghasilkan nilai 2 + 4 = 6. Nilai 6 tersebut akan menjadi nilai variabel *b*.

X.2 Membuat dan Memanggil (*Invoke*) Method

Pada contoh :

```
b = F ( 2 , 4 )
```

dikatakan bahwa fungsi F dipanggil oleh method yang "membungkus" *statement* b = F (2 , 4), dan nilainya di-*copy* ke variabel *b.* Dalam terminologi pemrograman Java, "fungsi" disebut sebagai "method", dan aktivitas pemanggilan method dinamakan "invoke", sedangkan parameter yang dilewatkan ke dalam fungsi dinamakan "argumen".

Sebelum suatu method dapat dipanggil / di-*invoke*, maka terlebih dahulu harus dilakukan pendefinisian method pada definisi class.

Bentuk umum penulisan method adalah sebagai berikut :

```
[modifiers] return_type method_identifier ([arguments]){
            method_code_block;
}
```

di mana :

d. *[modifiers]* merepresentasikan keywords pada teknologi Java yang memodifikasi cara-cara penggunaan method. Contoh : *public, protected, private, static, final*.
e. *return_type* adalah tipe nilai yang akan dikembalikan oleh method yang akan digunakan pada bagian lain dari program. *Return_type* pada method sama dengan tipe data pada variabel. *Return_type* dapat merupakan tipe data primitif maupun tipe data referensi.
f. *method_identifier* adalah nama method.
g. *([arguments])*, merepresentasikan sebuah daftar variabel yang nilainya dilewatkan / dimasukkan ke method untuk digunakan oleh method. Bagian ini dapat tidak diisi, dan dapat pula diisi dengan banyak variabel.
h. *method_code_block*, adalah rangkaian pernyataan / *statements* yang dibawa oleh method.

X.2.1 Bentuk Dasar Method

Bentuk method yang paling dasar adalah :

a. Tidak mempunyai argumen.
b. Tidak mempunyai return_value.

Contoh bentuk method yang paling dasar adalah Contoh 10.1.

Contoh 10.1

```
1  public void konversiCelciusKeReaumur( ){
2     celcius = 78;
3     reaumur = 4/5*celcius ;
4     System.out.println("Nilai Celcius = "+ celcius);
5     System.out.println("Nilai Reaumur = "+reaumur);
6  }
```

Pada Contoh 10.1 diperlihatkan bahwa method *konversiCelciusKeReaumur()* tidak memiliki argumen, dan return-value-

nya adalah *void*, yang berarti, fungsi ini tidak mengembalikan nilai apa-apa. Method tersebut hanya melakukan pemberian nilai '78' kepada variabel *celcius* (diasumsikan telah dideklarasikan di luar method), dan melakukan perhitungan nilai variabel *reaumur* (variabel *reaumur* diasumsikan telah dideklarasikan di luar method).

EKSPERIMEN

1. Buatlah method lain yang mengkonversikan nilai Reaumur ke Celcius, Celcius ke Fahrenheit, dan Fahrenheit ke Celcius
2. Buatlah method bebas.

Contoh 10.2 merupakan satu contoh aplikasi yang memperlihatkan proses penyimpanan dan pengambilan pada pada sebuah Lumbung Padi. Pada class LumbungPadi, terdapat 3 *instance variable* / variabel anggota, yaitu :

a. persediaan, yang menyimpan data *banyaknya persediaan padi lumbung.*
b. padiDisimpan, yang menyimpan data *banyaknya tambahan padi yang disimpan ke dalam lumbung.*
c. padiDiambil, yang menyimpan data *banyaknya padi yang diambil dari lumbung.*

Class LumbungPadi mempunyai method-method :

a. hitungPersediaan( ), yang akan melakukan pembaharuan nilai variabel *persediaan* dengan rumus :

```
persediaan = padiDisimpan - padiDiambil
```

b. simpanPadi( ), yang akan melakukan pembaharuan nilai variabel *padiDisimpan* dengan rumus :

```
padiDisimpan = padiDisimpan + tambahanPadi
```

dengan *tambahanPadi* dibuat default, yaitu 100.

c. ambilPadi( ), yang akan melakukan pembaharuan nilai variabel *padiDiambil* dengan rumus :

```
padiDiambil = padiDiambil + beratPadiYangDiambil
```

dengan *beratPadiYangDiambil* = 50.

d. hitungPersediaanPadi( ), yang akan melakukan perhitungan jumlah persediaan padi di lumbung, dengan rumus :

```
persediaan = padiDisimpan – padiDiambil
```

e. cetakPersediaan( ), yang akan mencetak ke layar monitor jumlah padi yang tersimpan di lumbung.

Class *Petani*, mempunyai 2 *instance variables*, yaitu :

a. beratPanenan, yang memuat informasi tentang berat panenan yang dituai oleh petani.
b. lumbung, yang memuat informasi tentang lumbung yang akan dijadikan tempat menyimpan panenan oleh petani.

Class *Petani* juga mempunyai method-method, yaitu :

a. lakukanPanen(), yang memperbaharui nilai variabel *beratPanenan* menjadi 150.
b. simpanPanenanDiLumbung( ), yang akan menyimpan hasil panen ke lumbung
c. ambilPanenanDariLumbung( ), yang akan mengambil hasil panen dari lumbung.

Sedangkan class *KegiatanPanen* merupakan *main class*, yaitu class yang mendefinisikan method *main*. Pada class *KegiatanPanen* terdapat method *main* yang berisi pernyataan-pernyataan yang menggambarkan kegiatan panen. Pada baris ke-3 sampai ke-6 dilakukan deklarasi dan inisialisasi obyek-obyek lumbungDesaSukatani (class : LumbungPadi), pakBakri (class : Petani), dan daengBaso(class : Petani).

Pada baris ke-8 dan ke-9, dilakukan inisialisasi variabel *lumbung* pada obyek *pakBakri* dan *daengBaso* yang referensinya disamakan dengan referensi obyek *lumbungDesaSukatani.*

Pada baris ke-10, pak Bakri melakukan panen. Hal ini diimplementasikan dengan pemanggilan method *lakukanPanen ( )* milik obyek *pakBakri*. Kemudian pada baris ke-11, pak Bakri melakukan penyimpanan hasil panen, yang diimplementasikan dengan pemanggilan method *simpanPanenanDiLumbung( )* milik obyek *pakBakri.* Kemudian pada baris ke-12, pak Bakri melakukan pengambilan hasil panen, yang diimplementasikan dengan pemanggilan method *ambilPanenanDariLumbung ( )* milik obyek *pakBakri*.

Pada baris ke-14, Daeng Baso melakukan panen, yang diimplementasikan dengan pemanggilan method *lakukanPanen( )* milik obyek *daengBaso.* Kemudian pada baris ke-15, Daeng Baso melakukan penyimpanan hasil panen, yang diimplementasikan dengan pemanggilan method *simpanPanenandiLumbung( )* milik obyek *daengBaso*.

Contoh 10.2

```
    //file : LumbungPadi.java
 1  public class LumbungPadi{
 2      public int persediaan = 0;
 3      public int padiDisimpan = 0;
 4      public int padiDiambil = 0;
 5
 6      public void hitungPersediaan( ){
 7          persediaan = padiDisimpan – padiDiambil;
 8      }
 9
10      public void simpanPadi( ){
11          int tambahanPadi = 100;
12          padiDisimpan = padiDisimpan + tambahanPadi;
13      }
14
15      public void ambilPadi ( ){
16          int beratPadiYangDiambil = 50;
17          padiDiambil =
18                  padiDiambil + beratPadiYangDiambil;
19      }
20
21      public void hitungPersediaanPadi( ){
22          persediaan = padiDisimpan – padiDiambil;
23      }
24
25      public void cetakPersediaan( ){
26          hitungPersediaan( );
27          System.out.println( "Persediaan di lumbung =
28                                  "+persediaan);
29      }
30  }
```

```
    //file : Petani.java

 1  public class Petani{
 2      public int beratPanenan;
 3      public LumbungPadi lumbung = new LumbungPadi( );
 4      public void lakukanPanen ( ){
 5          beratPanenan = 150;
 6      }
 7
 8      public void simpanPanenanDiLumbung( ){
 9          lumbung.simpanPadi( );
10      }
11
12      public void ambilPanenanDariLumbung( ){
13          lumbung.ambilPadi( );
14      }
15  }
```

```
   //file KegiatanPanen.java

 1   public class KegiatanPanen{
 2        public static void main(String[ ] args){
 3            LumbungPadi lumbungDesaSukatani =
 4                                    new LumbungPadi ( );
 5            Petani pakBakri = new Petani( );
 6            Petani daengBaso = new Petani( );
 7
 8            pakBakri.lumbung = lumbungDesaSukaTani;
 9            daengBaso.lumbung = lumbungDesaSukaTani;
10            pakBakri.lakukanPanen( );
11            pakBakri.simpanPanenanDiLumbung( );
12            pakBakri.ambilPanenanDariLumbung( );
13
14            daengBaso.lakukanPanen( );
15            daengBaso.simpanPanenanDiLumbung( );
16
17            lumbungDesaSukatani.cetakPersediaan( );
18        }
19   }
```

EKSPERIMEN

3. Buatlah 1 obyek Petani lagi dan lakukan interaksi dengan obyek *lumbungDesaSukatani* (menyimpan padi, mengambil padi). Catatan : jangan lupa menginisialisasi variabel *lumbung* dari obyek baru ini, seperti pada baris ke-7 dan ke-8. Apa yang terjadi jika inisialisasi variabel *lumbung* ini tidak dilakukan ?
4. Buatlah 1 obyek LumbungPadi lagi dan lakukan interaksi dengan obyek-obyek petani yang sudah ada. Lalu cetaklah persediaan padi di lumbung padi baru ini. Catatlah hasilnya.

X.2.2 Memanggil Method dari Class yang Berbeda

Method pada suatu class dapat dipanggil (istilah lain : di-*invoke*) oleh class lain. Ketika suatu method dipanggil, maka baris-baris statement yang berada dalam *code block*-nya akan dieksekusi.

Pada class *Petani*, method *simpanPanenanDiLumbung ( )* didefinisikan sebagai berikut :

```
 8      public void simpanPanenanDiLumbung( ){
 9          lumbung.simpanPadi( );
10      }
```

Method *simpanPanenanDiLumbung( )* pada class *Petani* memiliki *code block* yang berisi pernyataan pemanggilan method *simpanPadi( )* milik obyek *lumbung*. Hal ini akan mengakibatkan *code block* pada method *simpanPadi* milik obyek *lumbung* akan dieksekusi. Untuk melihat bentuk eksekusinya, lihat class *LumbungPadi*, yang merupakan cetak-biru dari obyek *lumbung*.

Method *simpanPanenanDiLumbung( )* memanggil method milik obyek yang berasal dari class yang berbeda, yaitu *simpanPadi( )*. Dari baris ke-8 sampai ke-10 pada file *Petani.java*, dapat disimpulkan bahwa untuk memanggil method yang berasal dari class yang berbeda, syntax-nya adalah sebagai berikut :

```
< object identifier > . < method identifier ([arguments])> ;
```

di mana :

a. *object identifier* adalah nama obyek
b. *method identifier* adalah nama method. Nama method harus merupakan nama method yang terdefinisi pada class yang menjadi cetak biru dari obyek.
c. *arguments* adalah nama argumen-argumen. Komposisi argumen pada method yang dipanggil, harus sama dengan komposisi argumen pada definisi method pada class yang menjadi cetak biru dari obyek.

X.2.3 Memanggil Method dari Class yang Sama

Perhatikan class *LumbungPadi* pada method *cetakPersediaan()*.

```
25       public void cetakPersediaan( ){
26           hitungPersediaan( );
27           System.out.println( "Persediaan di lumbung =
28           "+persediaan);
29       }
```

Pada method *cetakPersediaan( )* terdapat pemanggilan method *hitungPersediaan( )*. Method *persediaan ( )* berada pada class yang sama dengan method *cetakPersediaan ( )*, yaitu pada baris ke-6 sampai ke-8. Syntax untuk memanggil method yang berasal dari class yang sama, berbeda dengan pemanggilan method yang berasal dari class yang berbeda. Syntax-nya lebih sederhana ;

```
<method identifier([arguments])>;
```

di mana :

a. *method identifier* adalah nama method. Nama method harus merupakan nama method yang terdefinisi pada class yang menjadi cetak biru dari obyek.
b. *arguments* adalah nama argumen-argumen. Komposisi argumen pada method yang dipanggil, harus sama dengan komposisi argumen pada definisi method pada class yang menjadi cetak biru dari obyek.

X.3 Melewatkan Argumen dan Mengembalikan Nilai

Pada Contoh 10.2, untuk proses penyimpanan padi pada lumbung, jumlah padi yang akan disimpan tidak dapat ditentukan dari method yang memanggil. Ketika method *simpanPadi( )* dipanggil, terlebih dahulu variabel *tambahanPadi* diberi nilai 100 sebelum pembaharuan nilai variabel *padiDisimpan* dilakukan.

Agar lebih fleksibel (nilai *tambahanPadi* dapat ditentukan dari method yang memanggil method *simpanPadi( )*), maka dapat digunakan *argument* sebagai variabel *dummy,* atau parameter pada definisi method.

Contoh 10.3 merupakan modifikasi dari Contoh 10.2, di mana terdapat beberapa method yang memiliki argumen. Pada class *LumbungPadi*, terdapat method-method ber-argumen, yaitu :

a. simpanPadi(int tambahanPadi)
b. ambilPadi(int beratPadiYangDiambil)

Sedangkan pada class *Petani*, method-method berargumennya adalah :

a. simpanPanenanDiLumbung(int jumlahPanenan)
b. ambilPanenanDariLumbung(int panenanDiambil)

Dengan adanya method dengan argumen, pada method main (), banyaknya padi yang akan disimpan / diambil dapat ditentukan dengan lebih bebas. Contohnya adalah class *KegiatanPanen* pada Contoh 10.3 baris ke-11, 12 dan 15.

Contoh 10.3

```
    //file : LumbungPadi.java

1   public class LumbungPadi{
2       public int persediaan = 0;
3       public int padiDisimpan = 0;
4       public int padiDiambil = 0;
5
6       public void hitungPersediaan( ){
7           persediaan = padiDisimpan - padiDiambil;
8       }
9
```

```
10      public void simpanPadi(int tambahanPadi){
11          padiDisimpan = padiDisimpan + tambahanPadi;
12      }
13
14      public void ambilPadi (int beratPadiYangDiambil){
15          padiDiambil =
16                 padiDiambil + beratPadiYangDiambil;
17      }
18
19      public int  hitungPersediaanPadi( ){
20          persediaan = padiDisimpan – padiDiambil;
21          return persediaan;
22      }
23
24      public void cetakPersediaan( ){
25          int persediaanPadiTerakhir =
26                            hitungPersediaanPadi( );
27          System.out.println( "Persediaan di lumbung =
28                          " + persediaanPadiTerakhir);
29      }
30  }
```

```
    //file : Petani.java

 1  public class Petani{
 2      public int beratPanenan;
 3      public LumbungPadi lumbung = new LumbungPadi( );
 4      public void lakukanPanen ( ){
 5          beratPanenan = 150;
 6      }
 7
 8      public void simpanPanenanDiLumbung(int
 9                                     jumlahPanenan ){
10            lumbung.simpanPadi(jumlahPanenan);
11      }
12
13      public void ambilPanenanDariLumbung(int
14                                     panenanDiambil){
15           lumbung.ambilPadi( panenanDiambil);
16      }
17  }
```

```
    //file KegiatanPanen.java

 1  public class KegiatanPanen{
 2      public static void main(String[ ] args){
 3          LumbungPadi lumbungDesaSukatani =
 4                                new LumbungPadi ( );
 5          Petani pakBakri = new Petani( );
 6          Petani daengBaso = new Petani( );
 7
 8          pakBakri.lumbung = lumbungDesaSukatani;
```

```
 9           daengBaso.lumbung = lumbungDesaSukatani;
10           pakBakri.lakukanPanen();
11           pakBakri.simpanPanenanDiLumbung(100 );
12           pakBakri.ambilPanenanDariLumbung( 10);
13
14           daengBaso.lakukanPanen( );
15           daengBaso.simpanPanenanDiLumbung(90);
16
17           lumbungDesaSukatani.cetakPersediaan( );
18       }
19   }
```

EKSPERIMEN

5. Buatlah beberapa obyek Petani baru dan beberapa obyek LumbungPadi baru, dan buatlah interaksi antara obyek-obyek Petani dan LumbungPadi sebanyak mungkin. Cetaklah persediaan masing-masing lumbung padi.

X.3.1 Mendeklarasikan Method ber-Argumen

Deklarasi method ber-*argumen* dilakukan sesuai dengan syntax berikut :

```
[modifiers] return_type method_identifier (
        data_type argument_identifier_1 [,
        data_type argument_identifier_n]  ) {

                code_block;

 }
```

dengan :

a. *[modifiers]* merepresentasikan keywords pada teknologi Java yang memodifikasi cara-cara penggunaan method. Contoh : *public, protected, private, static, final.*
b. *return_type* adalah tipe nilai yang akan dikembalikan oleh method yang akan digunakan pada bagian lain dari program. *Return_type* pada method sama dengan tipe data pada variabel. *Return_type* dapat merupakan tipe data primitif maupun tipe data referensi.
c. *method_identifier* adalah nama method.
d. *data_type* merupakan tipe data dari argumen.

e. *([argument_identifier_n])*, merepresentasikan nama argumen ke-n.
f. *method_code_block*, adalah rangkaian pernyataan / *statements* yang dibawa oleh method.

Contoh deklarasi method dengan argumen dapat dilihat pada class *LumbungPadi* pada Contoh 10.3 baris ke-14 sampai ke-17 :

```
14        public void ambilPadi (int beratPadiYangDiambil){
15           padiDiambil =
16               padiDiambil + beratPadiYangDiambil;
17       }
```

X.3.2 Memanggil Method ber-Argumen

Memanggil method ber-argumen, baik dari class yang sama maupun class yang berbeda, sama saja caranya dengan memanggil method yang telah dibahas sebelumnya. Perbedaannya adalah, pada pemanggilan method ber-argumen, perlu ditentukan nilai yang akan dilewatkan ke method tersebut (*passing value).*

Nilai yang dilewatkan ke method tersebut harus mempunyai tipe data yang sama dengan tipe data argumen yang telah didefinisikan pada definisi class. Sebagai contoh adalah pemanggilan method *ambilPadi()* milik obyek *lumbung* pada method *ambilPanenanDariLumbung( ) milik* class *Petani* pada Contoh 10.3.

Pada baris ke-15, method *ambilPadi( )* milik obyek *lumbung* dipanggil dengan memasukkan nilai variabel *panenanDiambil* sebagai nilai yang dilewatkan ke method *ambilPadi( ).* Pernyataan ini valid karena variabel *panenanDiambil* merupakan argumen dari method *ambilPanenanDariLumbung()* yang bertipe integer.

```
13  public void ambilPanenanDariLumbung(int
14                                    panenanDiambil){
15      lumbung.ambilPadi( panenanDiambil);
16  }
```

X.3.3 Mendeklarasikan Method yang Memiliki Nilai Pengembalian

Method yang telah kita pelajari sampai saat ini adalah method yang tidak mengembalikan nilai apapun, yang ditandai dengan return_value *void*. Kita dapat membuat method yang mengembalikan nilai, dengan menggunakan return_value berupa :

a. salah satu dari tipe data primitif, atau
b. tipe data referensi

Untuk mendeklarasikan method yang memiliki nilai pengembalian, syntax-nya adalah sebagai berikut :

```
[modifiers] return_value method_identifier([arguments]){

        method_code_block;
        return value;

}
```

dengan :

a. *[modifiers]* merepresentasikan keywords pada teknologi Java yang memodifikasi cara-cara penggunaan method. Contoh : *public, protected, private, static, final*.

b. *return_type* adalah tipe nilai yang akan dikembalikan oleh method yang akan digunakan pada bagian lain dari program. *Return_type* dapat merupakan tipe data primitif maupun tipe data referensi.

c. *method_identifier* adalah nama method.

d. *([arguments])*,merepresentasikan sebuah daftar variabel yang nilainya dilewatkan / dimasukkan ke method untuk digunakan oleh method. Bagian ini dapat tidak diisi, dan dapat pula diisi dengan banyak variabel.

e. *method_code_block*, adalah rangkaian pernyataan / *statements* yang dibawa oleh method.

f. *return* adalah kata kunci pada teknologi Java untuk menyatakan bahwa method mengembalikan sebuah nilai , yaitu *value*.

g. *value*, yaitu sebuah nilai yang akan dikembalikan oleh method ke method yang memanggilnya.

Contoh deklarasi method yang mengembalikan nilai adalah pada method *hitungPersediaanPadi( )* pada class *LumbungPadi* pada Contoh 10.3.

```
19      public int  hitungPersediaanPadi( ){
20         persediaan = padiDisimpan - padiDiambil;
21         return persediaan;
22      }
```

Pada contoh di atas diperlihatkan bahwa nilai yang dikembalikan adalah nilai dari variabel *persediaan*, yang bertipe-data *int* (sama dengan *return_type* pada method.

X.3.4 Menerima Nilai Pengembalian

Nilai yang dihasilkan oleh method akan dikembalikan kepada method yang memanggilnya. Pada class *LumbungPadi* di Contoh 10.3,

method *cetakPersediaan( )* melakukan pemanggilan method *hitungPersediaanPadi( )* yang merupakan milik class *LumbungPadi* itu sendiri. Method *hitungPersediaanPadi( )* mengembalikan nilai *int* kepada method *cetakPersediaan( ).* Nilai *int* yang dikembalikan tersebut di-*copy* ke variabel lokal *persediaanPadiTerakhir*.

```
24        public void cetakPersediaan( ){
25            int persediaanPadiTerakhir =
26                                        hitungPersediaanPadi( );
27            System.out.println( "Persediaan di lumbung = "+
28                                      persediaanPadiTerakhir);
29        }
```

X.3.5 Keuntungan Menggunakan Method

Menggunakan method memiliki beberapa keuntungan, yaitu :

a. Method membuat program lebih mudah dibaca dan mudah untuk dipelihara / di-*maintain*. Misalnya pada class *LumbungPadi*, method *cetakPersediaan( )* akan lebih mudah dimengerti daripada :

```
int persediaanPadiTerakhir = hitungPersediaanPadi(
);
System.out.println( "Persediaan di lumbung = "+
      persediaanPadiTerakhir);
```

   Anggap saja, pada class *KegiatanPanen* tidak dilakukan pemanggilan method, tetapi menjabarkan semua kegiatan pada method-method tersebut, seperti pada Contoh 10.4. Ketika pemrogram ingin melakukan perbaikan pada kegiatan pengambilan padi yang dilakukan oleh obyek *pakBakri*, dia harus menganalisa dulu baris-baris mana saja yang termasuk dalam kegiatan yang ingin diperbaiki.

   Kesulitan ini akan teratasi jika pemrogram mengelompokkan pernyataan-pernyataan pada Contoh 10.4 menjadi method-method seperti pada Contoh 10.3.

b. Method membuat proses pengembangan dan perawatan (*maintenance*) menjadi lebih cepat. Jika pemrogram ingin melakukan penambahan kegiatan yang dapat dilakukan pada class *Petani*, maka pemrogram dapat membuat sebuah method dan menjabarkan proses-proses yang dibutuhkan dalam method tersebut. Ketika tiba waktunya *maintenance,* proses perbaikan akan terasa lebih cepat dibandingkan dengan, misalnya, menjabarkan proses-proses tanpa membuat method dengan menuliskan pernyataan-pernyataan pada method *main ( )*.

c. Method merupakan dasar untuk melakukan membuat software yang *re-usable*. Dapat terjadi, ada program lain yang membutuhkan method-method pada class *Petani*.

Program itu dapat menggunakan method-method tersebut setiap kali membutuhkan.

d. Method memungkinkan obyek-obyek yang berbeda untuk berkomunikasi dan untuk mendistribusikan beban kerja yang dipikul oleh program. Sebuah Obyek dapat berkomunikasi dengan obyek lain dengan cara melakukan modifikasi terhadap variabel anggota obyek lain, atau mengakses (mengambil nilai) variabel anggota obyek lain.

Contoh 10.4

```
    //file KegiatanPanen.java

    public class KegiatanPanen{
 1      public static void main(String[ ] args){
 2          LumbungPadi lumbungDesaSukatani =
 3                                    new LumbungPadi ( );
 4          Petani pakBakri = new Petani( );
 5          Petani daengBaso = new Petani( );
 6
 7          pakBakri.lumbung = lumbungDesaSukatani;
 8          daengBaso.lumbung = lumbungDesaSukatani;
 9
10          pakBakri.beratPanenan=150;
11          int jumlahPanenan = 100;
12          pakBakri.lumbung.padiDisimpan=
13          pakBakri.lumbung.padiDisimpan + jumlahPanenan;
14          int panenanDiambil = 10;
15          pakBakri.lumbung.padiDiambil=
16          pakBakri.lumbung.padiDiambil + panenanDiambil;
17
18          daengBaso.beratPanenan=150;
19          jumlahPanenan = 90;
20          daengBaso.lumbung.padiDisimpan=
21           daengBaso.lumbung.padiDisimpan+ jumlahPanenan;
22
23          lumbungDesaSukatani.persediaan=
24                lumbungDesaSukatani.padiDisimpan-
25                lumbungDesaSukatani.padiDiambil;
26          int persediaanPadiTerakhir =
27                     lumbungDesaSukatani . persediaan;
28          System.out.println( "Persediaan di lumbung = "+
29                              persediaanPadiTerakhir);
30          }
31  }
32
33
```

EKSPERIMEN

6. Buatlah tambahan aktivitas pada method *main ( )* tanpa menggunakan method :
   a. *pakBakri* panen
   b. *pakBakri* menyimpan padi di *lumbungDesaSukatani*
   c. *pakBakri* mengambil padi di *lumbungDesaSukatani*
   d. *daengBaso* mengambil padi di *lumbungDesaSukatani*
7. Buatlah modifikasi :
   a. Setiap kali panen, *pakBakri* hanya mendapatkan 30 kilogram
   b. Setiap kali panen, *daengBaso* mendapatkan 60 kilogram
   c. Setiap kali mengambil padi dari lumbung, *pakBakri* mengambil 20 kilogram, sedangkan *daengBaso* mengambil 15 kilogram.

Apakah memperbaiki program yang tidak menggunakan method dirasakan cukup mudah ?

X.4 Menggunakan Overloading pada Method

Suatu class dapat mengandung beberapa method dengan nama yang sama tetapi dengan komposisi argumen yang berbeda. Perhatikan class *Petani* pada Contoh 10.5.

Ada sedikit perubahan pada Contoh 10.5. Perhatikan pada class *KegiatanPanenan*. Sekarang didefinisikan terdapat 2 buah obyek *LumbungPadi*, yaitu :

a. *lumbungDesaSukatani*
b. *lumbungDesaSukamaju*

Perubahan lainnya adalah : obyek *pakBakri* mempunyai lumbung *default*, yaitu *lumbungDesaSukamaju*, sedangkan obyek *daengBaso* mempunyai lumbung *default*, yaitu *lumbungDesaSukatani*.

Perubahan berikutnya adalah pada class *Petani.* Pada class *Petani*, terdapat 3 buat method *simpanPanenanDiLumbung( )*, yaitu :

a. *simpanPanenanDiLumbung ( ),* yang akan melakukan proses penambahan nilai variabel *padiDisimpan* pada lumbung padi *default* sebesar 50 (kilogram).
b. *simpanPanenanDiLumbung (int jumlahPanenan)*, yang akan melakukan proses penambahan nilai variabel *padiDisimpan* pada lumbung padi *default* sebesar nilai yang dilewatkan ke method (nilai variabel *jumlahPanenan*).
c. *simpanPanenanDiLumbung (int jumlahPanenan, LumbungPadi lb)*, yang akan melakukan proses penambahan nilai variabel *padiDisimpan* pada lumbung padi *lb* (belum tentu merefer ke lumbung padi *default*).

Perhatikan class *KegiatanPanenan*. Pada method *main( )*, proses interaksi antara obyek *pakBakri* dengan lumbung-lumbung *lumbungPadiSukatani* dan *lumbungPadiSukamaju* diperlihatkan pada baris ke-12 sampai baris ke-15. Proses interaksi antara obyek *daengBaso* dengan kedua lumbung tersebut diperlihatkan pada baris ke-19.

Proses pada tiap baris digambarkan pada Tabel 10.1.

Tabel 10.1.
Proses pada Contoh 10.4

Baris ke-	lumbungPadi Sukatani		lumbungPadi Sukamaju	
	padiDisimpan	padiDiambil	padiDisimpan	padiDiambil
13	*0*	*0*	*10*	*0*
14	*0*	*0*	*60*	*0*
15	*40*	*0*	*60*	*0*
16	*40*	*0*	*60*	*10*
19	*130*	*0*	*60*	*10*

Nilai variabel *persediaan* pada obyek *lumbungPadiSukatani* adalah (130 – 0) = 130. Sedangkan nilai variabel *persediaan* pada obyek *lumbungPadiSukamaju* adalah (60 – 10) = 50. Oleh karena itu, ketika perintah cetak dieksekusi (baris ke-20 dan baris ke-21), maka outputnya adalah :

```
Persediaan di lumbung = 130
Persediaan di lumbung = 50
```

Contoh 10.5

```
//file : LumbungPadi.java

 1  public class LumbungPadi{
 2     public int persediaan = 0;
 3     public int padiDisimpan = 0;
 4     public int padiDiambil = 0;
 5
 6     public void hitungPersediaan( ){
 7       persediaan = padiDisimpan - padiDiambil;
 8     }
 9
10     public void simpanPadi(int tambahanPadi){
11       padiDisimpan = padiDisimpan + tambahanPadi;
12     }
13
14     public void ambilPadi (int beratPadiYangDiambil){
15       padiDiambil = padiDiambil + beratPadiYangDiambil;
16     }
17
18     public int  hitungPersediaanPadi( ){
19       persediaan = padiDisimpan - padiDiambil;
20       return persediaan;
21     }
22
23     public void cetakPersediaan( ){
24       int persediaanPadiTerakhir = hitungPersediaanPadi(
25  );
26       System.out.println( "Persediaan di lumbung = "+
27                               persediaanPadiTerakhir);
28     }
    }

//file : Petani.java

 1  public class Petani{
 2    public int beratPanenan;
 3    public LumbungPadi lumbung;
 4    public void lakukanPanen ( ){
 5      beratPanenan = 150;
 6    }
 7
 8    public void  simpanPanenanDiLumbung(){
 9       lumbung.simpanPadi(50);
10    }
11
12    public void simpanPanenanDiLumbung(int
13                                    jumlahPanenan ){
14         lumbung.simpanPadi(jumlahPanenan);
15    }
```

```
16     public void simpanPanenanDiLumbung(int jumlahPanenan,
17            LumbungPadi lb){
18        lb.simpanPadi(jumlahPanenan);
19     }
20
21     public void ambilPanenanDariLumbung(int
22                                            panenanDiambil){
23          lumbung.ambilPadi( panenanDiambil);
24     }
25   }
```

```
//file : KegiatanPanen.java
```

```
 1   public class KegiatanPanen{
 2     public static void main(String[ ] args){
 3        LumbungPadi lumbungDesaSukatani =
 4                                  new LumbungPadi ( );
 5        LumbungPadi lumbungDesaSukamaju =
 6                                  new LumbungPadi( );
 7        Petani pakBakri = new Petani( );
 8        Petani daengBaso = new Petani( );
 9        pakBakri.lumbung = lumbungDesaSukamaju;
10
11        daengBaso.lumbung = lumbungDesaSukatani;
12        pakBakri.lakukanPanen();
13        pakBakri.simpanPanenanDiLumbung(10);
14        pakBakri.simpanPanenanDiLumbung();
15        pakBakri.simpanPanenanDiLumbung(40,
16                                     lumbungDesaSukatani);
17        pakBakri.ambilPanenanDariLumbung( 10);
18        daengBaso.lakukanPanen( );
19        daengBaso.simpanPanenanDiLumbung(90);
20        lumbungDesaSukatani.cetakPersediaan( );
21        lumbungDesaSukamaju.cetakPersediaan( );
22     }
23   }
```

X.4.1 Overloading Method pada Java API

Pada Java API terdapat method-method yang di-overload. Misalnya pada class PrintStream, terdapat *overloading method* antara lain :

a. *print(boolean b)*
b. *print(char c)*
c. *print(char[ ] s)*
d. *print(double d)*
e. *print(float f)*
f. *print(int i)*
g. *print(long l)*
h. *print(Object o)*
i. *print(String s)*

11 VARIABEL DAN METHOD STATIK

XI.1 Membuat Method dan Variabel *static*

Variabel static adalah variabel yang dalam penggunaannya bukan menjadi milik eksklusif dari suatu class / object tertentu, meskipun definisinya berada pada suatu class.

Contoh 11.1 merupakan modifikasi dari Contoh 10.3, di mana semua variabel dan method pada class *LumbungPadi* mempunyai modifier *static*, yang berarti merupakan variabel dan method static. Hal ini berarti variabel-variabel dan method-method tersebut bukan milik eksklusif suatu obyek *LumbungPadi*.

Modifikasi lain pada Contoh 11.1 adalah ditiadakannya variabel anggota *lumbung* dari class *Petani*, sehingga untuk method-method yang perlu mengakses method dari class *LumbungPadi*, tidak mengambilnya dari obyek *LumbungPadi*, tetapi langsung memanggil method static.

Contoh 11.1

```
     //file : LumbungPadi.java
 1   public class LumbungPadi{
 2        public static  int persediaan = 0;
 3        public static int padiDisimpan = 0;
 4        public static int padiDiambil = 0;
 5
 6        public static void hitungPersediaan( ){
 7             persediaan = padiDisimpan - padiDiambil;
 8        }
 9
10        public static void simpanPadi(int tambahanPadi){
11             padiDisimpan = padiDisimpan + tambahanPadi;
12        }
13
14        public static void ambilPadi (int
15                                  beratPadiYangDiambil){
16             padiDiambil =
17                       padiDiambil + beratPadiYangDiambil;
18        }
19
```

```
20        public static int  hitungPersediaanPadi( ){
21            persediaan = padiDisimpan – padiDiambil;
22            return persediaan;
23        }
24
25        public static void cetakPersediaan( ){
26            int persediaanPadiTerakhir =
27                              hitungPersediaanPadi( );
28            System.out.println( "Persediaan di lumbung =
29                            " + persediaanPadiTerakhir);
30        }
31    }
```

```
    //file : Petani.java

 1  public class Petani{
 2       public int beratPanenan;
 3       public void lakukanPanen ( ){
 4           beratPanenan = 150;
 5       }
 6
 7       public void simpanPanenanDiLumbung(int
 8                                  jumlahPanenan ){
 9           LumbungPadi.simpanPadi(jumlahPanenan);
10       }
11
12       public void ambilPanenanDariLumbung(int
13                                    panenanDiambil){
14           LumbungPadi.ambilPadi( panenanDiambil);
         }
    }
```

```
    //file KegiatanPanen.java

 1  public class KegiatanPanen{
 2       public static void main(String[ ] args){
 3           Petani pakBakri = new Petani( );
 4           Petani daengBaso = new Petani( );
 5           pakBakri.lakukanPanen();
 6           pakBakri.simpanPanenanDiLumbung(100 );
 7           pakBakri.ambilPanenanDariLumbung( 10);
 8
 9           daengBaso.lakukanPanen( );
10           daengBaso.simpanPanenanDiLumbung(90);
11           LumbungPadi.cetakPersediaan( );
12       }
13  }
```

EKSPERIMEN

1. Buatlah sebuah obyek LumbungPadi pada class *KegiatanPanen*, lalu cetaklah persediaan dari obyek LumbungPadi yang Anda buat. Bandingkan dengan hasil yang dicetak oleh pernyataan pada class *KegiatanPanen* baris ke-14 Contoh 11.1. Mengapa hasilnya demikian ?
2. Tambahlah interaksi antara para petani dengan LumbungPadi pada class *KegiatanPanen*, lalu cetaklah persediaan padi pada obyek padi yang Anda buat pada eksperimen no. 1. Bandingkan dengan hasil yang dicetak oleh pernyataan pada class *KegiatanPanen* baris ke-14 Contoh 11.1. Mengapa hasilnya demikian ?

XI.1.1 Mendeklarasikan Method *static*

Method static dideklarasikan di dalam class. Meskipun demikian, method static bukan merupakan method yang menjadi milik eksklusif dari sebuah obyek.

Syntax pendeklarasian method static adalah sebagai berikut :

```
[modifiers] static return_type method_identifier(
        [arguments ]){

        method_code_block;
}
```

di mana :

a. *[modifiers]* merepresentasikan keywords pada teknologi Java yang memodifikasi cara-cara penggunaan method. Contoh : *public, protected, private, final*.
b. *static* adalah kata kunci pada teknologi Java yang menandakan bahwa method tersebut adalah method static. Catatan : *static* sebenarnya juga merupakan modifier, tetapi dalam syntax method static, *static* merupakan kata kunci yang wajib dituliskan.
c. *return_type* adalah tipe nilai yang akan dikembalikan oleh method yang akan digunakan pada bagian lain dari program. *Return_type* pada method sama dengan tipe data pada variabel. *Return_type* dapat merupakan tipe data primitif maupun tipe data referensi.
d. *method_identifier* adalah nama method.
e. *([arguments])*, merepresentasikan sebuah daftar variabel yang nilainya dilewatkan / dimasukkan ke method untuk

digunakan oleh method. Bagian ini dapat tidak diisi, dan dapat pula diisi dengan banyak variabel.

f. *method_code_block*, adalah rangkaian pernyataan / *statements* yang dibawa oleh method.

Contoh deklarasi method static adalah method *hitungPersediaanPadi( )* pada class *LumbungPadi* :

```
20      public static int  hitungPersediaanPadi( ){
21          persediaan = padiDisimpan - padiDiambil;
22          return persediaan;
23      }
```

Perbedaan deklarasi method static dengan method non-static hanyalah pada pada penambahan modifier wajib : *static*.

XI.1.2 Memanggil Method *static*

Pemanggilan / *invoking* method static berbeda dengan pemanggilan method non-static. Hal ini disebabkan oleh status method static yang bukan milik eksklusif dari obyek.

Karena status tersebut, maka pemanggilan method static mengikuti syntax berikut :

```
<class_name> . <method_identifier(arguments)> ;
```

di mana :

a. *class_name* adalah nama class yang mendefinisikan method static.

b. *method_identifier* adalah nama method.

c. *arguments* adalah argumen method static. Komposisi argumen harus sama dengan yang terdefinisi pada class.

Contoh pemanggilan method static adalah pada class *KegiatanPanen* pada baris ke-11 :

```
11          LumbungPadi.cetakPersediaan( );
```

XI.1.3 Mendeklarasikan Variabel *static*

Selain method static, teknologi Java juga memfasilitasi adanya variabel static. Sama dengan method static, variabel static juga merupakan variabel yang tidak menjadi milik eksklusif suatu obyek.

Syntax pendeklarasian dan inisialisasi method static adalah sebagai berikut :

```
[modifiers] static data_type identifier = value;
```

di mana :

a. *[modifiers]* merepresentasikan keywords pada teknologi Java yang memodifikasi cara-cara penggunaan variabel. Contoh : *public, protected, private, final*.
b. *static* adalah kata kunci pada teknologi Java yang menandakan bahwa variabel tersebut adalah variabel static.
c. *data_type* adalah tipe data, dapat berupa tipe data primitif, maupun tipe data referensi
d. *identifier* adalah nama variabel
e. *value* adalah nilai yang disimpan ke dalam variabel.

Contoh deklarasi dan inisialisasi variabel static, adalah pada class *LumbungPadi* pada baris ke-2 :

```
1  public class LumbungPadi{
2       public static  int persediaan = 0;
            ...
   }
```

XI.1.4 Method *static* dan Variabel *static* pada Java API

Java API (*Application Programming Interface*) mempunyai beberapa class yang mengandung method dan variabel static. Class-class tersebut diantaranya adalah :

a. System

Variabel static-nya antara lain :

e. public static PrintStream err;
f. public static InputStream in;
g. public static PrintStream out;

Method static-nya antara lain :

a) public static Properties getProperties( )
b) public static String getProperty(String key)
c) public static SecurityManager getSecurityManager( )
d) dan lain-lain

b. Math

Variabel static-nya antara lain :

a) public static double E;
b) public static double PI;

Method static-nya antara lain :

a) public static double abs (double a);
b) public static double cos (double a);
c) public static double exp (double a);
d) public static double max (double a, double b);
e) dan lain-lain

XI.1.5 Method *main ( )*

Method *main()* adalah satu method static khusus yang wajib ada dalam setiap aplikasi Java Standard Edition (kecuali untuk aplikasi Applet – tidak dibahas di modul ini). Method ini adalah method yang pertama kali dicari oleh Java Virtual Machine (JVM) ketika sebuah program dieksekusi. Jadi, agar program dapat dieksekusi, program tersebut harus mempunyai method *main ( )*.

Seperti method pada umumnya, method *main ( )* didefinisikan dalam sebuah class. Class yang mengandung method *main( )* disebut *main class*.

Syntax dari method *main( )* adalah sebagai berikut :

```
public static void main ( String[ ] args ){
        main_method_code_block ;
}
```

Method *main ( )* mempunyai argumen berupa array String *args*. Array String ini akan diisi nilai-nilai yang dimasukkan ketika program dipanggil. Sebagai contoh, ketika program *KegiatanPanen* dipanggil dengan menyertakan argumen :

```
C:> java KegiatanPanen  petani1  petani2  petani3
```

maka nilai dari semua variabel array args adalah :

args [0] = "petani1"
args [1] = "petani2"
args [2] = "petani3"

Catatan : Dalam modul Object-Oriented Programming 1 ini belum dibahas tentang array, jadi untuk sementara, tuliskan saja *String[ ] args* sebagai argumen dari method.

XI.1.6 Kapan Menggunakan Method atau Variabel *static* ?

Method atau variabel static dideklarasikan sebaiknya ketika :

a. Tidak diperlukan operasi pada obyek individual
b. Tidak diperlukan asosiasi suatu variabel kepada sebuah obyek
c. Diperlukan akses variabel atau method sebelum instanstiasi obyek
d. Method atau variabel secara logika bukan milik dari sebuah obyek, tetapi milik sebuah class utilitas. Contohnya class *Math* pada Java API

12 ENKAPSULASI DAN KONSTRUKTOR

XII.1 Menggunakan Enkapsulasi

Dalam pemrograman berorientasi obyek, istilah *enkapsulasi* berarti menyembunyikan data di dalam class. Data-data yang dimaksud adalah *instance variable* yang menjadi milik eksklusif dari class / obyek.

Sebenarnya, obyek dapat dikatakan sebagai entitas yang mengikat data-data menjadi data-data yang ekslusif. Pengikatan ini juga disebut *enkapsulasi* data.

XII.1.1 Visibility Modifier

Visibility Modifier adalah modifier-modifier yang memberi batasan kemampuan variabel atau method untuk diakses. Sebenarnya ada 4 buah modifier, yaitu *public, protected, default,* dan *private*. Hanya saja untuk Pemrograman Berbasis Obyek I hanya dibahas *public* dan *private*.

XII.1.2 Modifier *public*

Modifier *public* adalah modifier yang memberi kemampuan tak terbatas bagi variabel atau method untuk diakses. Artinya, variabel atau method yang menggunakan modifier *public* akan dapat diakses dari mana saja, baik dari dalam class sendiri, maupun dari class lain.

XII.1.3 Potensi Masalah dengan Atribut *public*

Selama ini kita telah membuat variabel dan method dengan modifier *public*. Permasalahan akan muncul ketika kita memberikan modifier *public* pada *instance variable /* variabel anggota. Perhatikan Contoh 12.1

Contoh 12.1

```
    // file : LumbungPadi2.java

 1  public class LumbungPadi2{
 2        public int persediaan;
 3        public int jumlahDiambil;
 4        public int jumlahDimasukkan;
```

```
 5
 6        public void cetakPersediaan( ){
 7             persediaan = persediaan + jumlahDimasukkan –
 8                   jumlahDiambil;
 9             System.out.println("Persediaan = " +
10                                        persediaan);
11        }
   }
```

```
   //file : KegiatanPanen2.java

 1 public class KegiatanPanen2{
 2       public static void main(String[ ] args){
 3             LumbungPadi2 lumbungDesaSukatani =
 4                                  new LumbungPadi2( );
 5             lumbungDesaSukatani.persediaan = 100;
 6             lumbungDesaSukatani.jumlahDimasukkan = 200;
 7             lumbungDesaSukatani.jumlahDiambil = 150;
 8             lumbungDesaSukatani.cetakPersediaan( );
 9       }
   }
```

Setelah mengkompilasi kedua file *.java pada Contoh 12.1, eksekusilah ! Output yang ditampilkan adalah :

```
Persediaan = -100
```

Pertanyaannya : Mungkinkan persediaan padi bernilai -100 ?

Ini adalah contoh masalah yang terjadi jika pada *instance variable* diterapkan modifier *public*. Semua class dapat melakukan modifikasi nilai variabel tanpa memperhatikan batasan-batasan yang seharusnya diberikan kepada variabel tersebut.

Secara logika, persediaan padi pada lumbung mempunyai nilai minimal=0. Desain class *LumbungPadi* seharusnya mengakomodasi pembatasan nilai tersebut.

Solusinya dapat dilihat pada bagian-bagian berikutnya dari bab ini.

XII.1.4 Modifier *private*

Selain *public*, terdapat modifier lain, yaitu *private*. Modifier ini membatasi aksesibilitas variabel atau method, sehingga hanya dapat diakses oleh variabel atau method dari class yang sama.

Perhatikan Contoh 12.2. Semua *instance variable* dari class *LumbungPadi2* diberi modifier *private*. Kemudian oleh method *main( )* pada class *KegiatanPanen2* variabel-variabel tersebut dimodifikasi nilainya.

Contoh 12.2

```
//file : LumbungPadi2.java

1  public class LumbungPadi2{
2        private  int persediaan;
3        private int jumlahDiambil;
4        private int jumlahDimasukkan;
5  }
```

```
//file : KegiatanPanen2.java

 1  public class KegiatanPanen2{
 2        public static void main(String[ ] args){
 3              LumbungPadi2 lumbungDesaSukatani =
 4                                     new LumbungPadi2( );
 5              lumbungDesaSukatani.persediaan = 100;
 6              lumbungDesaSukatani.jumlahDimasukkan =200;
 7              lumbungDesaSukatani.jumlahDiambil = 150;
 8              lumbungDesaSukatani.cetakPersediaan( );
 9        }
10  }
```

Jika class LumbungPadi2 dan KegiatanPanen2 dikompilasi, maka akan terjadi error :

```
--------------------------- Compiler Output ----------------
-----------
9_2\KegiatanPanen2.java:5: persediaan has private access in
LumbungPadi2 lumbungDesaSukatani.persediaan = 100;
                                ^
9_2\KegiatanPanen2.java:6:  jumlahDimasukkan  has  private
access  in  LumbungPadi2  lumbungDesaSukatani.jumlahDimasukkan
= 200;
                                ^
9_2\KegiatanPanen2.java:7:  jumlahDiambil has private access
in LumbungPadi2 lumbungDesaSukatani.jumlahDiambil = 150;
                                ^
9_2\KegiatanPanen2.java:8: cannot resolve symbol
symbol  : method cetakPersediaan ()
location:                class                 LumbungPadi2
lumbungDesaSukatani.cetakPersediaan();
                                                ^
4 errors
```

Hasil compilasi memperlihatkan bahwa method *main* ( ) tidak dapat memodifikasi nilai dari variabel *persediaan, jumlahDimasukkan, jumlahDiambil,* dan *cetakPersediaan* dari obyek *lumbungDesaSukatani.*

Ini buktu bahwa modifier *private* menyebabkan *instance variable* dan method tidak dapat diakses dari luar class / obyek tersebut.

Inilah yang disebut enkapsulasi ! Data yang disimpan oleh obyek *lumbungDesaSukatani* tidak dapat diakses oleh obyek di luar *lumbungDesaSukatani*.

XII.1.5 Interface dan Impementasinya

Contoh 12.2 memperlihatkan bagaimana dengan modifier *private*, data milik obyek terlindungi dari akses-akses dari obyek luar. Lalu bagaimana mengakses dan memodifikasi data tersebut ?

Untuk dapat mengakses data yang dilindungi pada obyek, diperlukan sebuah *interface* / antarmuka. Interface tersebut berupa method. Dengan menggunakan method, variabel anggota dapat diakses, dimodifikasi, dan dibatasi modifikasinya.

Salah satu kegunaan method sebagai interface untuk variabel-variabel anggota adalah : method dapat berfungsi sebagai *filter* untuk membatasi nilai variabel-variabel anggota pada class, seperti yang diperlihatkan pada Contoh 12.3.

Contoh 12.3

//file : LumbungPadi2.java

```
 1  public class LumbungPadi2{
 2          private  int persediaan;
 3          private int jumlahDiambil;
 4          private int jumlahDimasukkan;
 5
 6
 7      public void setPersediaan( int p ){
 8        System.out.println("SET PERSEDIAAN PADI");
 9         if( p>= 0){
10              persediaan = p;
11              System.out.println("Persediaan diset =
12                                      "+p+"\n");
13         }
14         else{
15              System.out.println ("Persediaan padi harus
16                            nol atau positif " ) ;
17         }
18         System.out.println("======================\n");
19      }
20
21      public int getPersediaan( ){
22         return persediaan;
23      }
24
```

```
25      public void setJumlahDiambil( int jdb ){
26         System.out.println("PENGAMBILAN PADI");
27         if( jdb < 0 ){
28             System.out.println ("Jumlah padi yang " +
29             "diambil harus nol atau positif ");
30
31         }else{
32           if( jdb <= persediaan ){
33               jumlahDiambil = jdb;
34               persediaan -= jumlahDiambil;
35               System.out.println("Jumlah Padi Diambil = "+
36                   jumlahDiambil);
37               System.out.println("Persediaan padi = "+
38                 persediaan);
39           }else{
40                 System.out.println("Tidak tersedia " +
41                     "padi dalam jumlah yang diinginkan ");
42                 System.out.println ("Jumlah padi yang "+
43                     "diinginkan = " +jumlahDiambil);
44                 System.out.println("Persediaan = " +
45                     persediaan);
46           }
47         }
48         System.out.println("=======================\n");
49      }
50
51      public int getJumlahDiambil( ){
52         return jumlahDiambil;
53      }
54
55      public void cetakPersediaan( ){
56        System.out.println("CETAK PERSEDIAAN PADI DI
57                                     LUMBUNG");
58        System.out.println("Persediaan padi = " +
59                                 persediaan);
60         System.out.println("=======================\n");
61
62      }
63
64      public void setJumlahDimasukkan( int jdm ){
65        String pesan = "SET JUMLAH PADI YANG DIMASUKKAN " +
66            "KE LUMBUNG";
67         System.out.println(pesan);
68         boolean kondisi = jdm>=0;
69         if( kondisi ){
70               jumlahDimasukkan = jdm;
71               persediaan  += jumlahDimasukkan;
72               System.out.println("Jumlah Padi Dimasukkan =
73                         "+ jumlahDimasukkan);
74               System.out.println("Persediaan padi = "+
75                 persediaan);
76         }
77         else{
```

```
78              System.out.println("Jumlah     padi     yang
79          dimasukkan "+
80                 "ke lumbung harus sama dengan nol " +
81                 "atau positif");
82          }
83          System.out.println("========================\n");
84      }
85
86      public int getJumlahDimasukkan( ){
87          return jumlahDimasukkan;
        }
    }
```

```
    //file : KegiatanPanen2.java

 1  public class KegiatanPanen2{
 2          public static void main(String[ ] args){
 3            LumbungPadi2 lumbungDesaSukatani =
 4                                      new LumbungPadi2( );
 5
 6            lumbungDesaSukatani.setPersediaan(100);
 7            lumbungDesaSukatani.setJumlahDimasukkan(100);
 8            lumbungDesaSukatani.setJumlahDiambil(250);
 9            lumbungDesaSukatani.cetakPersediaan();
10
11          }
12  }
```

Contoh 12.3 memperlihatkan bagaimana method-method pada class *LumbungPadi* melakukan pemeriksaan dahulu terhadap variabel yang akan dimodifikasi nilainya. Misalnya pada method *setJumlahDiambil ( )*. Method ini mempunyai sebuah argumen integer. Method ini digunakan untuk memodifikasi variabel *jumlahDiambil* pada class *LumbungPadi.*

Tetapi sebelum memodifikasi variabel *jumlahDiambil* tersebut, method ini melakukan langkah-langkah mengujian, yaitu :

a. Menguji apakah nilai argumen yang dilewatkan bernilai negatif (baris ke-26 class *LumbungPadi2*). Jika bernilai negatif, maka akan keluar pesan kesalahan, dan eksekusi akan keluar dari method.
b. Jika nilai argumen yang dilewatkan bernilai positif, maka dilakukan pengujian apakah besarnya lebih kecil atau sama dengan persediaan (baris ke-32). Jika benar, maka nilai dari argumen akan di-*copy* ke variabel *jumlahDiambil*. Kemudian variabel *persediaan* diperbaharui dengan rumus (baris ke-34) :

```
persediaan = persediaan - jumlahDiambil;
```

c. Jika nilai argumen bernilai lebih besar dari variabel *persediaan* , maka program akan mengeluarkan pesan kesalahan (baris ke-39 sampai baris ke-46). Hal ini dilakukan karena secara logika, tidak mungkin permintaan pengambilan padi melebihi persediaan yang ada.

Dengan penggunaan interface berupa method , maka method pemanggil (*caller*) tidak perlu melakukan pemeriksaan terhadap variabel, karena hal itu telah dilakukan oleh method pekerja (*worker*).

Setelah Contoh 12.3 dikompilasi dan dijalankan, maka output program adalah sebagai berikut :

```
SET PERSEDIAAN PADI
Persediaan diset = 100
========================

SET JUMLAH PADI YANG DIMASUKKAN KE LUMBUNG
Jumlah Padi Dimasukkan = 100
Persediaan padi = 200
========================

PENGAMBILAN PADI
Tidak tersedia padi dalam jumlah yang diinginkan
Jumlah padi yang diinginkan = 250
Persediaan = 200
========================

CETAK PERSEDIAAN PADI DI LUMBUNG
Persediaan padi = 200
========================
```

Terlihat bahwa ketika dilakukan pengambilan padi dengan jumlah padi lebih besar daripada persediaan, maka program akan melakukan tindakan pemberitahuan bahwa jumlah padi yang ingin diambil tidak dapat dipenuhi oleh lumbung padi. Dengan demikian variabel anggota dapat dikendalikan nilainya, dan tidak menyimpang dari desain.

EKSPERIMEN

1. Ubahlah baris ke-6 sampai ke-8 sebagai berikut :

```
lumbungDesaSukatani.setPersediaan(20);
lumbungDesaSukatani.setJumlahDimasukkan(100);
lumbungDesaSukatani.setJumlahDiambil(80);
```

   Kompilasi dan jalankan program tersebut. Outputnya bagaimana ? Mengapa outputnya seperti itu ?

2. Ubahlah baris ke-6 sampai ke-8 sebagai berikut :

```
lumbungDesaSukatani.setPersediaan(-100);
lumbungDesaSukatani.setJumlahDimasukkan(-100);
lumbungDesaSukatani.setJumlahDiambil(-250);
```

   Kompilasi dan jalankan program tersebut. Outputnya bagaimana ? Mengapa outputnya seperti itu ?

XII.2 Mendeskripsikan *Variable Scope*

Variable scope adalah ruang lingkup keteraksesan variabel. Pendefinisian scope menentukan pada bagian mana saja suatu variabel dapat diakses. Berdasarkan scope-nya, variabel pada teknologi Java dibagi atas 2 macam :

a. *instance variable*, yaitu variabel yang dapat digunakan pada semua bagian obyek. Misalnya, variabel tersebut dapat diakses oleh method pada obyek tersebut. Variabel ini adalah variabel anggota pada class, yang dideklarasikan di dalam class tetapi di luar method.
b. *local variable,* yaitu variabel yang hanya dapat digunakan pada method yang mendeklarasikannya.

Sebagai contoh, method *setJumlahDimasukkan ( )* pada Contoh 12.3 mempunyai 2 buah *local variable*, yaitu :

a. *pesan* (String)
b. *kondisi* (boolean)

Kedua variabel lokal tersebut dideklarasikan di dalam method *setJumlahDimasukkan ( )* sehingga penggunaannya terbatas hanya di

dalam method tersebut. Dengan kata lain jika method lain mengakses kedua variabel lokal tersebut, maka ketika program dikompilasi, maka akan terjadi error.

EKSPERIMEN

3. Buatlah sebuah method baru untuk mengakses variabel-variabel lokal dan anggota, sebagai berikut :

```
public void aksesVariabel( ){
     //mengakses variabel "jumlahDiambil"
     System.out.println("Jumlah padi yang diambil =
                        " + jumlahDiambil);
     //mengakses variabel "pesan" milik
    System.out.println("Bunyi pesan pada method " +
                setJumlahDimasukkan adalah " + pesan);
}
```

Panggillah method ini pada method *main( )*

XII.2.1 Penempatan *Instance Variable* dan *Local Variable* pada Memori

Pada memory, *Instance Variable* disimpan pada Heap Memory. Sedangkan *Local Variable* Stack Memory.

Pada *main class* pada Contoh 12.3, terdapat instanstiasi obyek *LumbungPadi* yang diberi identifier *lumbungDesaSukatani.* Pada Stack Memory, variabel referensi *lumbungDesaSukatani* ditempatkan pada *scope* yang diberi nama *main*. Lalu variabel-variabel instans / *instance variable* milik *lumbungDesaSukatani* dialokasikan pada Heap Memory dan dikelompokkan menjadi *obyek LumbungPadi*. Variabel referensi yang berada di Stack Memory akan berisi alamat memory dari obyek *LumbungPadi* tersebut.

Lalu pada method *main( )*, method *setPersediaan( )* milik *lumbungDesaSukatani* dipanggil. Pada definisi class *LumbungPadi*, method *setPersediaan ( )* memiliki argumen *p*. Karena argumen *p* ini merupakan variabel lokal milik method *setPersediaan( )* maka pada Stack Memory dibuatlah *scope* dengan nama *setPersediaan.* Argumen *p* diletakkan pada *scope* tersebut, dan diberi nilai 100 karena nilai yang dilewatkan ke method *setPersediaan( )* adalah 100.

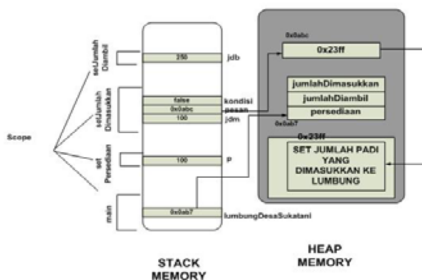
Kemudian method *main ( )* memanggil method *setJumlahDimasukkan* ( ) milik obyek *lumbungDesaSukatani*. Pada definisi class *LumbungPadi* , terdapat argumen *jdm* pada method

setJumlahDimasukkan, kemudian pada *code_block* method tersebut terdapat instanstiasi obyek *pesan*, diikuti dengan deklarasi dan inisialisasi variabel *kondisi*. Karena variabel-variabel *jdm, pesan,* dan *kondisi* merupakan variabel lokal milik method *setJumlahDimasukkan*, maka variabel-variabel tersebut ditempatkan pada *scope* pada Stack Memory yang diberi nama *setJumlahDimasukkan*.

Method *main( )* kemudian memanggil method *setJumlahDiambil( )* milik obyek *lumbungDesaSukatani*. Method ini mempunyai argumen *jdb*, dan tidak menambahkan variabel lokal pada *code_block*-nya. Oleh karena itu, pada Stack Memory, dibuatlah sebuah *scope* yang diberi nama *setJumlahDiambil*, yang diisi variabel lokal *jdb*. Variabel lokal ini diberi nilai 250.

Terakhir, method *main( )* memanggil method *cetakPersediaan()* milik *lumbungDesaSukatani.* Method ini tidak membuat variabel lokal apapun, sehingga pada Stack Memory tidak dibuat *scope*.

Penempatan instans variabel dan local variabel pada memory dapat dilihat pada Gambar 12.1.



Gambar 12.1 Penempatan Instance Variable dan Local Variable pada Memori

XII.3. Konstruktor

Konstruktor adalah struktur yang mirip dengan method (tetapi bukan method), yang digunakan untuk melakukan instanstiasi obyek. Ketika dipanggil, Konstruktor akan melakukan inisialisasi nilai variabel-variabel anggota / atribut dari obyek yang akan diinstanstiasi.

XII.3.1 Mendefinisikan Konstruktor

Konstruktor dapat didefinisikan melalui definisi class. Syntax umumnya adalah sebagai berikut :

```
[modifiers] class className {
     [modifiers]  ConstructorName([arguments])
{
               code_block;

     }
}
```

di mana :

a. *[modifiers]* merepresentasikan keywords pada teknologi Java yang memodifikasi aksesibilitas terhadap konstruktor. Contoh : *public, protected,* atau *private.*

b. *ConstructorName* adalah nama konstruktor. *ConstructorName* harus sama dengan *className*.

c. *([arguments])*,merepresentasikan sebuah daftar variabel yang nilainya dilewatkan / dimasukkan ke konstruktor. Bagian ini dapat tidak diisi, dan dapat pula diisi dengan banyak variabel.

d. *code_block*, adalah rangkaian pernyataan / *statements* yang dibawa oleh constructor.

Catatan : Perhatikan bahwa pada pendefinisian konstruktor, tidak digunakan *return_type* seperti yang biasa digunakan pada pendefinisian method.

Pendefinisian konstruktor dapat dilihat pada Contoh 12.4.

Contoh 12.4.

```
//file : LumbungPadi2.java

 1  public class LumbungPadi2{
 2        private int persediaan;
 3        private int jumlahDiambil;
 4        private int jumlahDimasukkan;
 5
 6      public LumbungPadi2(){
 7          persediaan = 100;
 8          jumlahDiambil = 10;
 9          jumlahDimasukkan = 20;
10      }
11  }
```

Pada Contoh 12.4, terdapat pendefinisian konstruktor *LumbungPadi2()*. Ketika konstruktor ini dipanggil, maka terjadi modifikasi terhadap variabel instans : *persediaan*=100, *jumlahDiambil*=10, dan *jumlahDimasukkan*=20. Jadi ketika instanstiasi selesai, obyek *LumbungPadi2* telah memiliki nilai awal untuk masing-masing variabel instans-nya.

XII.3.2 Konstruktor Default

Konstruktor default pada semua class adalah konstruktor yang tidak memiliki argumen. Konstruktor default tidak perlu didefinisikan. Jadi, jika sebuah class tidak mendefinisikan konstruktor, maka ketika obyek diinstanstiasi, maka konstruktor default dapat digunakan.

XII.3.3 Konstruktor Overloading

Sebuah class dapat memiliki lebih dari satu konstruktor. Sama seperti method, setiap konstruktor tidak dapat memiliki komposisi argumen yang sama. Seperti diperlihatkan pada Contoh 12.5.

Contoh 12.5.

```
     //file : LumbungPadi2.java

 1   public class LumbungPadi2{
 2          private int persediaan;
 3          private int jumlahDiambil;
 4          private int jumlahDimasukkan;
 5
 6        public LumbungPadi2(){
 7            persediaan = 0;
 8            jumlahDiambil = 0;
 9            jumlahDimasukkan = 0;
10        }
11
12        public LumbungPadi2(int psd ){
13            persediaan = psd;
14            jumlahDiambil = 0;
15            jumlahDimasukkan = 0;
16        }
17
18        public LumbungPadi2(int psd, int jdimasukkan){
19            persediaan = psd;
20            jumlahDiambil = 0;
21            jumlahDimasukkan = jdimasukkan;
22        }
23   }
```

Contoh 12.5 memperlihatkan class *LumbungPadi2* yang memiliki 3 buah konstruktor :

a. *LumbungPadi2()*
b. *LumbungPadi2(int)*
c. *LumbungPadi2(int,int)*

Dalam eksekusi program, jika sebuah konstruktor dipanggil, maka *code block* konstruktor bersangkutan akan dijalankan. Perhatikan Contoh 12.6.

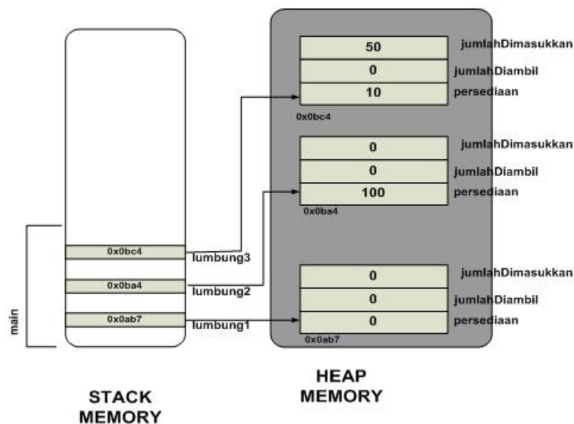
Contoh 12.6

```
   //file : KegiatanPanen2.java
1  public class KegiatanPanen2{
2       public static void main (String[] args){
3            LumbungPadi2 lumbung1 = new LumbungPadi2();
4            LumbungPadi2 lumbung2 = new LumbungPadi2(100);
5            LumbungPadi2 lumbung3 = new LumbungPadi2 (10,50);
6       }
7  }
```

Pada Contoh 12.6 diinstanstiasi 3 obyek *LumbungPadi2,* yaitu *lumbung1, lumbung2,* dan *lumbung3*. Pada instanstiasi *lumbung1* (baris ke-3), konstruktor *LumbungPadi2()* dipanggil. Menurut konstruktor pada Contoh 12.6, maka semua variabel instan pada *lumbung1* (*persediaan, jumlahDiambil,* dan *jumlahDimasukkan*) bernilai 0.

Sedangkan pada instanstiasi *lumbung2* (baris ke-4), konstruktor *LumbungPadi2(int)* dipanggil. Sesuai dengan definisi konstruktor pada Contoh 12.6, maka isi variabel *persediaan* adalah 100, sedangkan *jumlahDiambil*, dan *jumlahDimasukkan* bernilai 0.

Pada instanstiasi *lumbung3* (baris ke-5), konstruktor *LumbungPadi2(int,int)* dipanggil. Sesuai dengan definisi konstruktor pada Contoh 12.6, maka isi variabel *persediaan* adalah 10, *jumlahDiambil adalah 0,* dan *jumlahDimasukkan* adalah 50. Visualisasi Memorinya dapat dilihat pada Gambar 12.2.



Gambar XII.2 Visualisasi Memori untuk 3 Obyek LumbungPadi

DAFTAR PUSTAKA

1. Horton Ivor, *Beginning Java 2*, Wrox Press, 2000, ISBN 1861003668
2. Naughton Patrick, *Java Hand Book*, McGraw-Hill, 2000, ISBN 9795334700
3. Sun Academic Initiative, *Fundamentals of Java Programming Language SL-110*, Sun Microsystem Press, 2005.