BAB 1
Struktur Kontrol

1.1 Tujuan

Pada bab sebelumnya, kita sudah mendapatkan contah dari program *sequential*, dimana *statement* dieksekusi setelah *statement* sebelumnya dengan urutan tertentu. Pada bagian ini, kita mempelajari tentang struktur kontrol yang bertujuan agar kita dapat menentukan urutan *statement* yang akan dieksekusi.

Pada akhir bab, siswa diharapkan mampu:

- Menggunakan struktur kontrol keputusan *(if, else, switch)* yang digunakan untuk memilih blok kode yang akan dieksekusi
- Menggunakan struktur kontrol pengulangan *(while, do-while, for)* yang digunakan untuk melakukan pengulangan pada blok kode yang akan dieksekusi
- Menggunakan *statement* percabangan *(break, continue, return)* yang digunakan untuk mengatur *redirection* dari program

1.2 Struktur Kontrol Keputusan

Struktur kontrol keputusan adalah *statement* dari *Java* yang mengijinkan *user* untuk memilih dan mengeksekusi blok kode dan mengabaikan blok kode yang lain.

1.2.1 Statement if

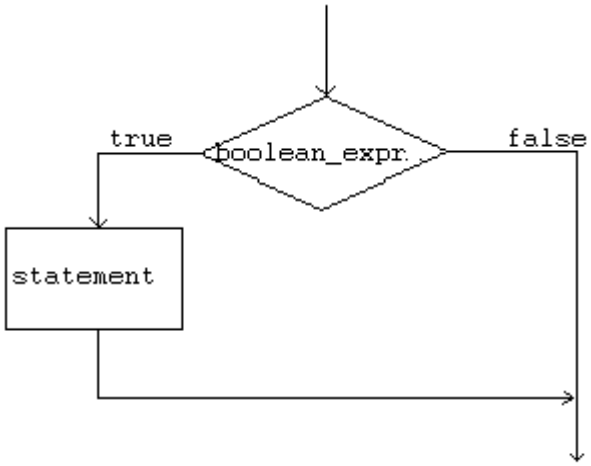
Statement-if menentukan sebuah *statement* (atau blok kode) yang akan dieksekusi jika dan hanya jika persyaratan boolean (*boolean statement*) bernilai *true.*

Bentuk dari *statement if*,

```
if( boolean_expression )
       statement;
```

atau

```
if( boolean_expression ){
       statement1;
       statement2;
       . . .
}
```



Gambar 1: Flowchart Statement If

dimana, *boolean_expression* adalah sebuah persyaratan *boolean* (*boolean statement*) atau *boolean* variabel.

Berikut ini adalah contoh *code statement if*,

```
int grade = 68;

if( grade > 60 ) System.out.println("Congratulations!");
```

atau

```
int grade = 68;

if( grade > 60 ){
       System.out.println("Congratulations!");
       System.out.println("You passed!");
}
```

Petunjuk Penulisan Program :

*1. **Boolean_expression** pada statement harus merupakan nilai boolean.Hal ini berarti persyaratan harus bernilai **true** atau **false**.*
2. Masukkan statement di dalam blok if. Contohnya,

```
if( boolean_expression ){
//statement1;
//statement2;
}
```

1.2.2 Statement if-else

Statement if-else digunakan apabila kita ingin mengeksekusi sebuah *statement* dengan kondisi *true* dan *statement* yang lain dengan kondisi *false*.

Bentuk *statement if-else*,

```
if( boolean_expression )
       statement;
else
       statement;
```

dapat juga ditulis seperti,

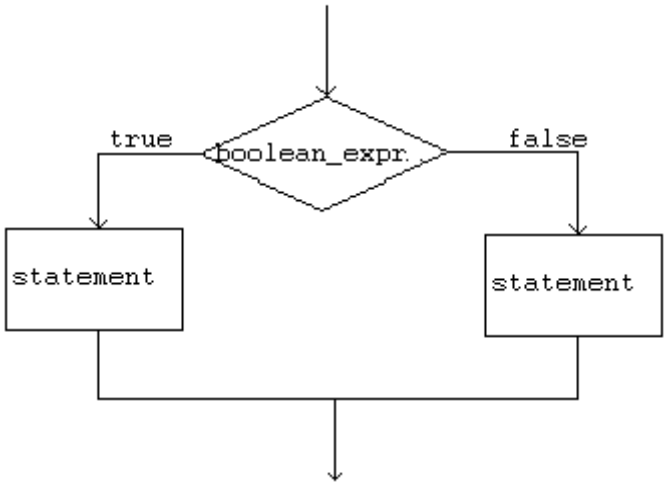
```
if( boolean_expression ){
       statement1;
       statement2;
       . . .
}
else{
       statement1;
       statement2;
       . . .
}
```

Berikut ini contoh *code statement if-else*,

```
int grade = 68;

if( grade > 60 )   System.out.println("Congratulations!");
else            System.out.println("Sorry you failed");
```

atau

```
int grade = 68;

if( grade > 60 ){
       System.out.println("Congratulations!");
       System.out.println("You passed!");
}
else{
       System.out.println("Sorry you failed");
}
```



Gambar 2: Flowchart Statement If-Else

Petunjuk Penulisan Program :

1. *Untuk menghindari kebingungan, selalu letakkan statement di dalam blok if-else di dalam tanda {},*
2. *Anda dapat memiliki blok if-else yang bersarang. Ini berarti anda dapat memiliki blok if-else yang lain di dalam blok if-else. Contohnya,*

```
if( boolean_expression ){
        if( boolean_expression ){
                . . .
        }
}
else{
 . . .
}
```

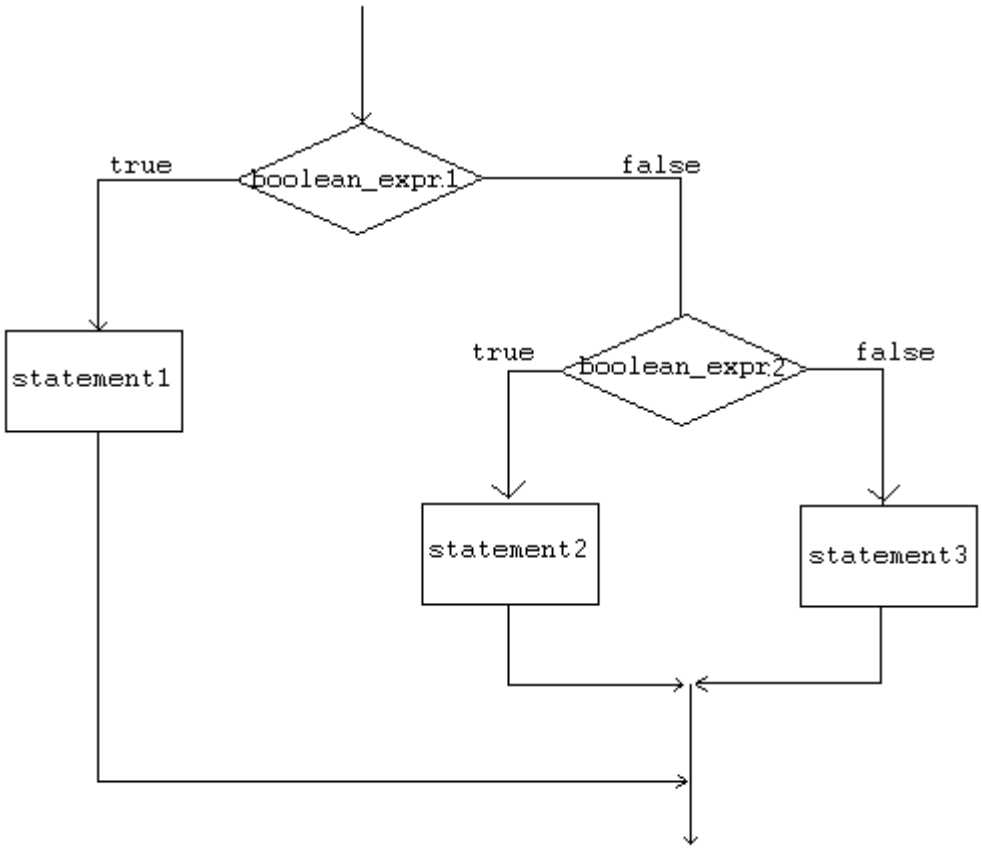
1.2.3 Statement if-else-if

Statement pada bagian *else* dari blok *if-else* dapat menjadi struktur *if-else* yang lain. Struktur seperti ini mengijinkan kita untuk membuat seleksi persyaratan yang lebih kompleks.

Bentuk *statement if-else if,*

```
if( boolean_expression1 )
       statement1;
else if( boolean_expression2 )
       statement2;
else
       statement3;
```

Bisa anda catat anda dapat memiliki banyak blok *else-if* sesudah *statement if*. Blok *else* bersifat optional dan dapat dihilangkan. Pada contoh di bawah atas, jika *boolean_expression1* bernilai *true*, maka program akan mengeksekusi *statement1* dan melewati *statement* yang lain. Jika *boolean_expression2* bernilai *true*, maka program akan mengeksekusi *statement2* dan melewati *statement3*.



Gambar 3: Flowchart Statement If-Else-If

Berikut ini contoh *code statement if-else-if*

```
int grade = 68;

if( grade > 90 ){
	System.out.println("Very good!");
}
else if( grade > 60 ){
	System.out.println("Very good!");
}
else{
	System.out.println("Sorry you failed");
}
```

1.2.4 Kesalahan umum ketika menggunakan statement if-else:

1. Kondisi pada *statement if* bukan merupakan nilai boolean. Contohnya,

```
//BENAR
int number = 0;
if( number ){
	//some statements here
}
```

Variabel number tidak memiliki nilai Boolean.

2. *Using = instead of == for comparison. For example*,
3. Menggunakan = daripada == untuk operator perbandingan. Contohnya,

```
//SALAH
int number = 0;
if( number = 0 ){
	//Statement Selanjutnya
}
```

Seharusnya *code* tersebut ditulis,

```
//BENAR
int number = 0;
if( number == 0 ){
	//Statement Selanjutnya
}
```

3. Menulis ***elseif*** daripada ***else if***.

1.2.5 Contoh statement if-else-else if

```
public class Grade
{
    public static void main( String[] args )
    {
        double grade = 92.0;

        if( grade >= 90 ){
               System.out.println( "Excellent!" );
        }
        else if( (grade < 90) && (grade >= 80)){
               System.out.println("Good job!" );
        }
        else if( (grade < 80) && (grade >= 60)){
               System.out.println("Study harder!" );
        }
        else{
               System.out.println("Sorry, you failed.");
        }
    }
}
```

1.2.6 Statement switch

Cara lain untuk membuat percabangan adalah dengan menggunakan kata kunci ***switch***. Dengan menggunakan *switch* kita bisa melakukan percabangan dengan persyaratan yang beragam.

Bentuk *statement switch*,

```
switch( switch_expression ){
      case case_selector1:
               statement1;        //
               statement2;        //block 1
               . . .              //
               break;

      case case_selector2:
               statement1;        //
               statement2;        //block 2
               . . .              //
               break;
      . . .
      default:
               statement1;        //
               statement2;        //block n
               . . .              //
               break;
}
```

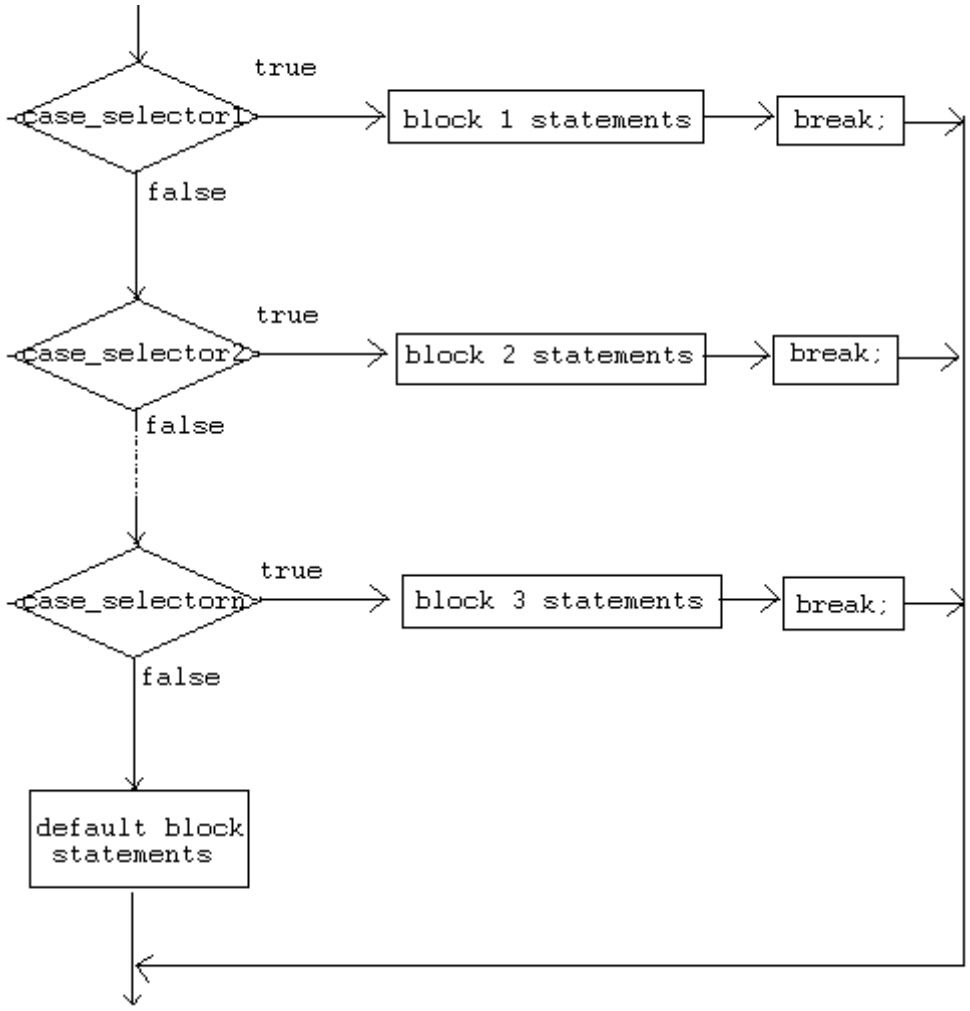
dimana, *switch_expression* adalah persyaratan ***integer*** atau ***character*** dan *case_selector1, case_selector2* dan seterusnya adalah konstanta nilai *integer* yang *unique* (unik).

Ketika *statement switch* ditemukan, pertama kali *Java* memeriksa *switch_expression*, dan meloncat ke *case* dan mencocokkan nilai yang sama dengan persyaratannya. Program mengeksekusi *statement* dari awal sampai menemui *statement break*, dan melewati *statement* yang lain sampai akhir struktur *switch*.

Jika tidak ditemui *case* yang cocok, maka program akan mengeksekusi blok *default*. Bisa anda catat bahwa blok *default* adalah optional. Sebuah *statement switch* bisa tidak memiliki blok default.

CATATAN:

- Tidak seperti *statement* ***if***, pada struktur *switch statement* dieksekusi tanpa memerlukan tanda kurung kurawal (**{}**).
- Ketika sebuah *case* pada *statement switch* menemui kecocokan, semua *statement* pada *case* tersebut akan dieksekusi. Tidak hanya demikian, *statement* lain yang berada pada *case* yang cocok juga dieksekusi.
- Untuk menghindari program mengeksekusi *statement* pada *case* berikutnya, kita menggunakan *statement* ***break*** sebagai *statement* akhir.



Gambar 4: Flowchart Statement Switch

Petunjuk Penulisan Program :

1. *Menentukan penggunaan statement if atau statement switch adalah sebuah keputusan. Anda dapat menentukan yang mana yang akan dipakai berdasarkan kemudahan membaca program dan faktor-faktor yang lain.*
2. *Statement if dapat digunakan untuk membuat keputusan berdasarkan rentang nilai tertentu atau kondisi tertentu, sedangkan statement switch membuat keputusan hanya berdasarkan nilai unique (unik) dari integer atau character.*

1.2.7 Contoh statement switch

```
public class Grade
{
    public static void main( String[] args )
    {
        int grade = 92;

        switch(grade){
       case 100:
               System.out.println( "Excellent!" );
               break;
       case 90:
               System.out.println("Good job!" );
               break;
       case 80:
               System.out.println("Study harder!" );
               break;
       default:
               System.out.println("Sorry, you failed.");
        }
    }
}
```

1.3 Struktur Kontrol Perulangan

Struktur kontrol pengulangan adalah *statement* dari *Java* dimana kita bisa mengeksekusi blok code berulang-ulang dalam kurun nilai tertentu. Ada tiga macam jenis struktur kontrol pengulangan yaitu *while, do-while*, dan *for-loops*.

1.3.1. while loop

Statement while loop adalah *statement* atau blok *statement* yang diulang-ulang sampai mencapai kondisi yang cocok.

Bentuk *statement while*,

```
while( boolean_expression ){
       statement1;
       statement2;
       . . .
}
```

Statement di dalam *while loop* akan dieksekusi berulang-ulang selama *boolean_expression* bernilai *true*.

Contoh, pada *code* dibawah ini,

```
int i = 4;
while ( i > 0 ){
       System.out.print(i);
       i--;
}
```

Contoh diatas akan mencetak angka 4321 pada layar. Perlu dicatat jika bagian i--; dihilangkan, akan menghasilkan *looping* yang tidak berhenti (**infinite loop**). Sehingga, ketika menggunakan *while loop* atau bentuk pengulangan yang lain, pastikan Anda memberikan *statement* yang membuat pengulangan berhenti pada suatu titik.

Berikut ini adalah beberapa contoh *while loop*,

Contoh 1:

```
int x = 0;
while (x<10)
{
       System.out.println(x);
       x++;
}
```

Contoh 2:

```
//infinite loop
while(true)
       System.out.println(“hello”);
```

Contoh 3:

```
//no loops
// statement is not even executed
while (false)
       System.out.println(“hello”);
```

1.3.2. do-while loop

Do-while loop mirip dengan *while-loop*. *Statement* di dalam *do-while loop* akan dieksekusi beberapa kali selama kondisi bernilai *true*.

Perbedaan antara while dan *do-while loop* adalah dimana *statement* di dalam *do-while loop* dieksekusi sedikitnya **satu kali.**

Bentuk *statement do-while,*

```
do{
        statement1;
        statement2;
        . . .
}while( boolean_expression );
```

Statement di dalam *do-while loop* akan dieksekusi pertama kali, dan dilakukan pengecekan kondisi dari *boolean_expression*. Jika nilai tersebut belum mencapai nilai yang diinginkan, *statement* akan dieksekusi lagi.

Berikut ini beberapa contoh *do-while loop*:

Contoh 1:

```
int x = 0;
do
{
        System.out.println(x);
        x++;
}while (x<10);
```

Contoh ini akan memberikan *output* 0123456789 pada layar.

Contoh 2:

```
//infinite loop
do{
         System.out.println(“hello”);
} while (true);
```

Contoh di atas akan melakukan pengulangan yang tidak berhenti untuk menulis *“hello”* pada layar.

Contoh 3:

```
//one loop
// statement is executed once
do
  System.out.println(“hello”);
while (false);
```

Contoh di atas akan memberikan *output hello* pada layar.

Panduan pemrograman:

1. Kesalahan pemrograman ketika menggunakan do-while loop adalah lupa untuk menulis titik koma (;) setelah ekspresi while.

```
do{
    ...
}while(boolean_expression)//-→ salah>tidak ada titik koma(;)
```

2. Seperti pada while loop, pastikan do-while loop anda berhenti pada suatu titik.

1.3.3. for loop

Seperti pada struktur pengulangan sebelumnya yaitu melakukan pengulangan eksekusi *code* beberapa kali.

Bentuk dari *for loop*,

```
for (InitializationExpression; LoopCondition; StepExpression){
      statement1;
      statement2;
      . . .
}
```

dimana,

InitializationExpression – inisialisasi dari variabel loop.
LoopCondition **-** membandingkan variabel loop pada nilai batas.
StepExpression - melakukan update pada variabel loop.

Berikut ini adalah contoh dari *for loop*,

```
int i;
for( i = 0; i < 10; i++ ){
      System.out.print(i);
}
```

Pada contoh ini, *statement* i=0 merupakan inisialisasi dari variabel. Selanjutnya, kondisi i<10 diperiksa. Jika kondisi bernilai *true, statement* di dalam *for loop* dieksekusi. Kemudian, *statement* i++ dieksekusi, dan dilakukan pengecekan kondisi. Kondisi ini akan dilakukan berulang-ulang sampai mencapai nilai yang salah *(false)*.

Contoh tadi, adalah contoh yang sama dari *while loop*,

```
int i = 0;
while( i < 10 ){
      System.out.print(i);
      i++;
}
```

1.4 Branching *Statements*

Branching statements mengijinkan kita untuk mengatur jalannya eksekusi program. *Java* memberikan tiga bentuk *branching statements: break, continue* dan *return*.

1.4.1 break statement

Statement break memiliki dua bentuk: *unlabeled* dan *labeled*.

1.4.1.1 Unlabeled break *statement*

Unlabeled menghentikan jalannya *statement switch*. Anda bisa juga menggunakan bentuk *unlabeled* untuk menghentikan *for, while* atau *do-while loop*.

Contohnya,

```
String names[] = {"Beah", "Bianca", "Lance", "Belle",
                  "Nico", "Yza", "Gem", "Ethan"};

String           searchName = "Yza";
boolean          foundName = false;

for( int i=0; i< names.length; i++ ){
       if( names[i].equals( searchName )){
                  foundName = true;
                  break;
       }
}

if( foundName ){
       System.out.println( searchName + " found!" );
}
else{
       System.out.println( searchName + " not found." );
}
```

Pada contoh ini, jika *string "Yza"* ditemukan, pengulangan pada *for loop* akan dihentikan dan akan melanjutkan ke proses berikutnya.

1.4.1.2 Labeled break *statement*

Bentuk *labeled form* dari *statement break* akan menghentikan *statement* luar, dimana diidentifikasikan berupa label pada *statement break*. Program berikut ini akan mencari nilai dalam *array* dua dimensi. Terdapat dua pengulangan bersarang *(nested loop)*. Ketika sebuah nilai ditemukan, *labeled break* akan menghentikan *statement* yang diberi label *searchLabel*, dimana label ini berada di luar.

```
int[][] numbers = {{1, 2, 3},
                   {4, 5, 6},
                   {7, 8, 9}};

int searchNum = 5;
boolean foundNum = false;

searchLabel:
for( int i=0; i<numbers.length; i++ ){
       for( int j=0; j<numbers[i].length; j++ ){
                  if( searchNum == numbers[i][j] ){
                            foundNum = true;
                            break searchLabel;
                  }
       }
}

if( foundNum ){
       System.out.println( searchNum + " found!" );
}
else{
       System.out.println( searchNum + " not found!" );
}
```

Statement break menghentikan sementera *labeled statement*; ia tidak lagi menjalankan *flow control* pada label. *Flow control* pada label akan di-*transfer* secara otomatis mengikuti *labeled statement*.

1.4.2 Continue statement

Statement continue memiliki dua bentuk: *unlabeled* dan *labeled*. Anda dapat menggunakan *statement continue* untuk melewati pengulangan dari *for, while*, atau *do-while loop* yang sedang berjalan.

1.4.2.1 Unlabeled continue *statement*

Bentuk *unlabeled* akan melewati akhir *statement* pada bagian yang dalam dan memeriksa *boolean expression* yang mengkontrol *loop*, pada dasarnya akan melewati bagian pengulangan pada *loop*.

Berikut ini adalah contoh dari penghitungan angka dari *"Beah"* dalam suatu *array*.

```
String names[] = {"Beah", "Bianca", "Lance", "Beah"};
int    count = 0;

for( int i=0; i<names.length; i++ ){

       if( !names[i].equals("Beah") ){
                continue; //skip next statement
       }

       count++;
}

System.out.println("There are " + count + " Beahs in the
list");
```

1.4.2.2 Labeled continue *statement*

Bentuk *labeled* akan melanjutkan sebuah *statement* dengan melewati pengulangan yang sedang berjalan dari *loop* terluar yang diberi label (tanda).

```
outerLoop:
for( int i=0; i<5; i++ ){

                                                    for( int j=0;
j<5; j++ ){


System.out.println("Inside for(j) loop"); //message1

                                                    if( j == 2 )
                                                    continue
                                                    }


System.out.println("Inside for(i) loop"); //message2
}
```

Pada contoh ini, pesan ke-2 tidak dicetak, karena *statement continue* akan melewati pengulangan yang sedang berjalan.

1.4.3 Return statement

Statement return digunakan untuk keluar dari sebuah fungsi *(method)*. *Statement return* memiliki dua bentuk: menggunakan sebuah nilai, dan tidak memberikan nilai.

Untuk memberikan sebuah nilai, cukup berikan nilai (atau ekspresi yang menghasilkan sebuah nilai) sesudah *return*. Contohnya,

```
return ++count;
```

atau

```
return "Hello";
```

Tipe data dari nilai yang diberikan harus sama dengan tipe dari fungsi yang dideklarasikan. Ketika sebuah *method void* dideklariskan, gunakan bentuk *return* yang tidak memberikan nilai. Contohnya,

```
return;
```

Kita akan membahas lebih lanjut tentang *statement return* ketika mempelajari tentang fungsi.

1.5 Latihan

1.5.1 Nilai

Ambil tiga nilai ujian dari *user* dan hitung nilai rata-rata dari nilai tersebut. Berikan *output* rata-rata dari tiga ujian. Berikan juga *smiley face* pada *output* jika nilai rata-rata lebih besar atau sama dengan 60, selain itu beri *output* :-(.

1. Gunakan *BufferedReader* untuk mendapat *input* dari *user*, dan *System.out* untuk *output* hasilnya.
2. Gunakan *JOptionPane* untuk mendapat *input* dari *user* dan *output* hasilnya.

1.5.2 Membaca Bilangan

Ambil sebuah angka sebagai *input* dari *user*, dan *outputnya* berupa kata yang sesuai dengan angka. Angka yang dimasukkan antara 1-10. Jika *user* memasukkan nilai yang tidak sesuai berikan *output "Invalid number"*.

1. Gunakan *statement if-else* untuk menyelesaikan
2. Gunakan *statement switch* untuk menyelesaikan

1.5.3 Cetak Seratus Kali

Buat sebuah program yang mencetak nama Anda selama seratus kali. Buat tiga versi program ini menggunakan *while loop, do while* dan *for-loop*.

1.5.4 Perpangkatan

Hitung pangkat sebuah nilai berdasarkan angka dan nilai pangkatnya. Buat tiga versi dari program ini menggunakan *while loop, do-while* dan *for-loop*.

BAB 2
Java Array

2.1 Tujuan
Dalam bagian ini, kita akan mendiskusikan mengenai *array* dalam *Java*. Pertama, kita akan mendefinisikan apa yang dimaksud dengan *array*, kemudian kita juga akan mendefinisikan bagaimana mendeklarasikannya dan menggunakannya dalam *Java*.

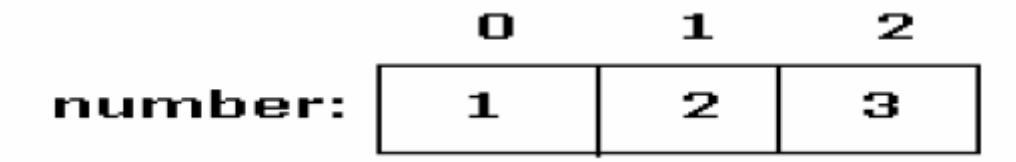
Pada akhir pelajaran, siswa haruslah mampu untuk :
- Mendeklarasikan dan membuat *array*
- Mengakses elemen-elemen didalam *array*
- Menentukan jumlah *element* didalam sebuah *array*
- Mendeklarasikan dan membuat *array* multidimensi

2.2 Pengenalan Array
Dibagian sebelumnya, kita telah mendiskusikan bagaimana cara pendeklarasian berbagai macam variabel dengan menggunakan tipe data primitif. Dalam pendeklarasian variabel, kita sering menggunakan sebuah tipe data beserta nama variabel atau *identifier* yang unik, dimana untuk menggunakan variabel tersebut, kita akan memanggil dengan nama *identifier*-nya.

Sebagai contoh, kita memiliki tiga variabel dengan tipe data int yang memiliki *identifier* yang berbeda untuk tiap variabel.

```
int number1;
int number2;
int number3;

number1 = 1;
number2 = 2;
number3 = 3;
```

Seperti yang dapat Anda perhatikan pada contoh diatas, hanya untuk menginisialisasi dan menggunakan variabel terutama pada saat variabel-variabel tersebut memiliki tujuan yang sama, dirasa sangat membingungkan. Di *Java* maupun di bahasa pemrograman yang lain, mereka memiliki kemampuan untuk menggunakan satu variabel yang dapat menyimpan sebuah data *list* dan kemudian memanipulasinya dengan lebih efektif. Tipe variabel inilah yang disebut sebagai ***array***.



Gambar 5: Contoh dari Integer Array

Sebuah *array* akan menyimpan beberapa *item* data yang memiliki tipe data sama didalam sebuah blok memori yang berdekatan yang kemudian dibagai menjadi beberapa *slot*. Bayangkanlah *array* adalah sebuah variabel – sebuah lokasi memori tertentu yang memiliki satu nama sebagai *identifier*, akan tetapi ia dapat menyimpan lebih dari sebuah *value*.

2.3 Pendeklarasian Array

Array harus dideklarasikan seperti layaknya sebuah variabel. Apabila Anda mendeklarasikan *array,* Anda harus membuat sebuah *list* dari tipe data, yang diikuti oleh tanda kurung buka dan kurung tutup, yang diikuti oleh nama *identifier*. Sebagai contoh,

```
int []ages;
```

atau Anda dapat menempatkan kurung buka dan kurung tutupnya setelah *identifier*. Sebagai contoh,

```
int ages[];
```

Setelah pendeklarasian, kita harus membuat *array* dan menentukan berapa panjangnya dengan sebuah **konstruktor**. Proses ini di *Java* disebut sebagai *instantiation* (Kata dalam *Java* yang berarti membuat). Untuk meng-*instantiate* sebuah obyek, kita

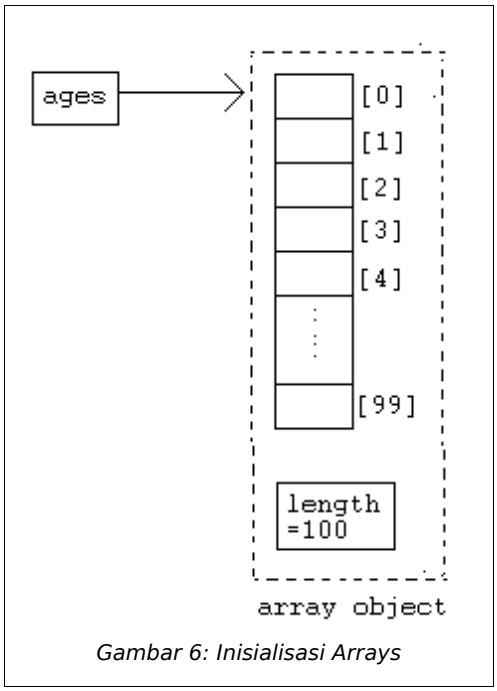
membutuhkan sebuah konstruktor. Kita akan membicarakan lagi mengenai *instantiate* obyek dan pembuatan konstruktor pada bagian selanjutnya. Perlu dicatat, bahwa ukuran dari *array* tidak dapat diubah setelah Anda menginisialisasinya. Sebagai contoh,

```
//deklarasi
int ages[];

//instantiate obyek
ages = new int[100];
```

atau bisa juga ditulis dengan,

```
//deklarasi dan instantiate
obyek
int ages[] = new
int[100];
```

Pada contoh diatas, deklarasi akan memberitahukan kepada *compiler Java*, bahwa *identifier ages* akan digunakan sebagai nama *array* yang berisi data-data *integer*, dan kemudian untuk membuat atau meng-*instantiate* sebuah *array* baru yang terdiri dari 100 elemen.



Gambar 6: Inisialisasi Arrays

Selain menggunakan sebuah *keyword* baru untuk meng-*instantiate array*, Anda juga dapat secara otomatis mendeklarasikan *array*, membangunnya, kemudian memberikan sebuah *value*.

Sebagai contoh,

```
//membuat sebuah array yang berisi variabel-variabel //boolean
pada sebuah identifier. Array ini terdiri dari 4 //elemen yang
diinisilisasikan sebagai value //{true,false,true,false}
boolean results[] ={ true, false, true, false };

//Membuat sebuah array yang terdiri dari penginisialisasian //4
variabel double bagi value {100,90,80,75}
double []grades = {100, 90, 80, 75};

//Membuat sebuah array String dengan identifier days. Array
//ini terdiri dari 7 elemen.
String days[] = { “Mon”, “Tue”, “Wed”, “Thu”, “Fri”, “Sat”,
“Sun”};
```

2.4 Pengaksesan sebuah elemen array

Untuk mengakses sebuah elemen dalam *array*, atau mengakses sebagian dari *array*, Anda harus menggunakan sebuah nomor atau yang disebut sebagai ***index*** atau *subscript*.
Sebuah **nomor *index* atau *subscript*** telah diberikan kepada tiap anggota *array*, sehingga program dan *programmer* dapat mengakses setiap *value* apabila dibutuhkan. Index **selalu dalam *integer***. **Dimulai dari nol, kemudian akan terus bertambah sampai *list value* dari *array* tersebut berakhir.** Perlu dicatat, bahwa elemen-elemen didalam *array* dimulai dari **0 sampai dengan (ukuran*Array*-1).**

Sebagai contoh, pada *array* yang kita deklarasikan tadi, kita mempunyai,

```
//memberikan nilai 10 kepada elemen pertama array
ages[0] = 10;

//mencetak elemen array yang terakhir
System.out.print(ages[99]);
```

Perlu diperhatikan bahwa sekali *array* dideklarasikan dan dikonstruksi, nilai yang disimpan dalam setiap anggota *array* akan diinisialisasi sebagai nol. Oleh karena itu, apabila Anda menggunakan tipe data *reference* seperti *String*, ia tidak akan diinisalisasi ke *string* kosong “”, sehingga Anda tetap harus membuat *String array* secara eksplisit.

Berikut ini adalah contoh, bagaimana untuk mencetak seluruh elemen didalam *array*. Dalam contoh ini digunakanlah *loop*, sehingga kode kita menjadi lebih pendek.

```
public class ArraySample{
	public static void main( String[] args ){

		int[] ages = new int[100];

		for( int i=0; i<100; i++ ){
			System.out.print( ages[i] );
		}
	}
}
```

Petunjuk penulisan program:
1. Biasanya, lebih baik menginisialisasi atau meng-instantiate array setelah Anda mendeklarasikannya. Sebagai contoh pendeklarasiannya

```
	int []arr = new int[100];
```
lebih disarankan daripada,
```
	int []arr;
	arr = new int[100];
```

2. Elemen-elemen dalam n-elemen array memiliki index dari 0 sampai n-1. Perhatikan disini bahwa tidak ada elemen array arr[n]. Hal ini akan menyebabkan array-index out-of-bounds exception.

3. Anda tidak dapat mengubah ukuran dari sebuah array

2.5 Panjang Array

Untuk mengetahui berapa banyak *element* didalam sebuah *array*, Anda dapat menggunakan ***length* (panjang)** *field* dalam *array*. Panjang *field* dalam *array* akan mengembalikan ukuran dari *array* itu sendiri. Sebagai contoh,

```
arrayName.length
```

Pada contoh sebelumnya, kita dapat menuliskannya kembali seperti berikut ini,

```
public class ArraySample
{
	public static void main( String[] args ){

		int[] ages = new int[100];

		for( int i=0; i<ages.length; i++ ){
			System.out.print( ages[i] );
		}
	}
}
```

Petunjuk penulisan program:

1. *Pada saat pembuatan loop untuk memproses elemen-elemen dalam array, gunakanlah length field didalam pernyataan pengkondisian dalam loop. Hal ini akan menyebabkan loop secara otomatis menyesuaikan diri terhada ukuran array yang berbeda-beda.*
2. *Pendeklarasian ukuran array di Java, biasanya digunakan constant untuk mempermudah. Sebagai contoh,*

```
final int ARRAY_SIZE = 1000;		//pendeklarasian constant
. . .

int[] ages = new int[ARRAY_SIZE];
```

2.6 Array Multidimensi

Array multidimensi diimplementasikan sebagai *array* didalam *array*. *Array multidimensi* dideklarasikan dengan menambahkan jumlah tanda kurung setelah nama *array*. Sebagai contoh,

```
// Elemen 512 x 128 dari integer array
int[][] twoD = new int[512][128];

// karakter array 8 x 16 x 24
char[][][] threeD = new char[8][16][24];

// String array 4 baris x 2 kolom
String[][] dogs = {{ "terry", "brown" },
                   { "Kristin", "white" },
                   { "toby", "gray"},
                   { "fido", "black"}
                   };
```

Untuk mengakses sebuah elemen didalam *array* multidimensi, sama saja dengan mengakses *array* satu dimensi. Misalnya saja, untuk mengakses *element* pertama dari baris pertama didalam *array dogs*, kita akan menulis,

```
System.out.print( dogs[0][0] );
```

Kode diatas akan mencetak *String* “terry” di layar.

2.7 Latihan

2.7.1 Hari dalam seminggu

Buatlah sebuah *String array* yang akan menginisialisasi 7 hari dalam seminggu. Sebagai contoh,

```
String days[] = {“Monday”, “Tuesday”….};
```

Gunakan *while-loop*, kemudian *print* semua nilai dari *array* (Gunakan juga untuk *do-while dan for-loop)Using a while-loop*.

2.7.2 Nomor terbesar

Gunakanlah *BufferedReader* dan *JoptionPane*, tanyakan kepada *user* untuk 10 nomor. Kemudian gunakan *array* untuk menyimpan 10 nomor tersebut. Tampilkan kepada *user*, *input* terbesar yang telah diberikan *user*.

2.7.3 Buku Alamat

Berikut ini adalah *array* multidimensi yang menyatakan isi dari sebuah buku alamat:

```
String entry = {{"Florence", "735-1234", "Manila"},
                {"Joyce", "983-3333", "Quezon City"},
                {"Becca", "456-3322", "Manila"}};
```

Cetak buku alamat tersebut dalam format berikut ini:

```
Name   : Florence
Tel. # : 735-1234
Address         : Manila

Name   : Joyce
Tel. # : 983-3333
Address         : Quezon City

Name   : Becca
Tel. # : 456-3322
Address         : Manila
```

BAB 3
Bekerja dengan Java Class Library

3.1 Tujuan

Pada sesi ini, kita akan mengantarkan beberapa konsep dasar dari *Object-Oriented objcts*, dan *Programming* (OOP). Selanjutnya kita akan membahas konsep dari *classes* dan bagaimana menggunakan *class* dan anggotanya. Perubahan dan pemilihan *object* juga akan dibahas. Sekarang, kita akan *focus* dalam menggunakan *class* yang telah dijabarkan dalam *Java Class library*, kita akan membahas nanti tentang bagaimana membikin *class* anda sendiri.

Pada akhir pelajaran, siswa seharusnya dapat :
1. menjelaskan OOP dan beberapa konsepnya
2. perbedaan antara *class* dan *object*
3. pebedaan antara *instance variables/method* dan *class (static) variable/method*
4. menjelaskan *method* apa dan bagaimana memanggil *method* parameter
5. mengidentifikasi beberapa jangkauan dari sebuah *variable*
6. memilih tipe data *primitive* dan *object*
7. membandingkan *objects* dan menjabarkan *class* dari *objects*.

3.2 Pengenalan Pemrograman Berorientasi Object

OOP berputar pada konsep dari *object* sebagai dasar element dari program anda. Ketika kita membandingkan dengan dunia nyata, kita dapat menemukan beberapa objek disekitar kita, seperti mobil, singa, manusia dan seterusnya. *Object* ini dikarakterisasi oleh sifat / attributnya dan tingkah lakunya.

Contohnya, objek sebuah mobil mempunyai sifat tipe transmisi, warna dan manufaktur. Mempunyai kelakuan berbelok, mengerem dan berakselerasi. Dengan cara yang sama pula kita dapat mendefinisikan perbedaan sifat dan tingkah laku dari singa. Coba perhatikan table dibawah ini sebagai contoh perbandingan :

Object	***Properties***	***Behavior***
Car	type of transmission manufacturer color	turning braking accelerating
Lion	Weight Color hungry or not hungry tamed or wild	roaring sleeping hunting

Table 1: Example of Real-life Objects

Dengan deskripsi ini, objek pada dunia nyata dapat secara mudah dimodelisasi sebagai

objek *software* menggunakan sifat sebagai data dan tingkah laku sebagai *method*. Data disini dan *method* dapat digunakan dalam *pemrograman game atausoftware* interaktif untuk menstimulasi objek dunia nyata. Contohnya adalah sebagai *software* objek mobil dalam permainan balap mobil atau *software* objek singadalam sebuah *software* pendidikan interaktif pada kebun binatang untuk anak anak.

3.3 Class dan Object

3.3.1 Perbedaan Class dan Object

Pada dunia *software*, sebuah objek adalah sebuah komponen *software* yang stukturnya mirip dengan objek pada dunia nyata. Setiap objek dibuat dari satu set data (sifat) dimana *variable* menjabarkan esensial karakter dari objek, dan juga terdiri dari satu set dari *methode* (tingkah laku) yang menjabarkan bagaimana tingkah laku dari objek. Jadi objek adalah sebuah berkas *software* dari *variable* dan *method* yg berhubungan. *Variable* dan *methods* dalam objek *Java* scara formal diketahui sebagai *instance variable* dan *instance methods* untuk membedakannya dari *variable* klas dan *method* klas, dimana akan dibahas kemudian.

Klas adalah sturktur dasar dari OOP. Dia terdiri dari dua tipe dari anggota dimana disebut dengan *field (attribut/properti)* dan *method. Field* mespesifikasi tipe data yang didefinisikan oleh *class*, sementara *methode* spesifikasi dari operasi. Sebuah objek adalah sebuah *instance* pada *class*.

Untuk dapat membedakanantara *class* dan *object*, mari kita mendiskusikan beberapa contoh. Apa yang kita miliki disini adalah sebuah *class* mobil dimana dapat digunakan untuk medefinisikan beberapa *object* mobil. Pada table dibawah, mobil A dan mobil B adalah objek dari kelas mobil. Kelas memiliki *field plat* nomer, warna, manufaktur, dan kecepatan yang diisi dengan nilai korespondendi pada objek mobil A dan mobil B. mobil juga dapat berakselerasi, berbelok dan mengerem.

Car Class		**Object Car A**	**Object Car B**
Instance Variables	Plate Number	ABC 111	XYZ 123
	Color	Blue	Red
	Manufacturer	Mitsubishi	Toyota
	Current Speed	50 km/h	100 km/h
Instance Methods	Accelerate Method		
	Turn Method		
	Brake Method		

Table 2: Contoh class car dan object-object nya

Ketika diinisialisi, tiap objek mendapat satu *set* baru dari *state variable*. Bagaimanapun, implementasi dari *method* dibagi diantara objek pada kelas yang sama.

Kelas menyediakan keuntungan dari *reusability*. *Software programmers* dapat digunakan dari sebuah kelas lagi dan lagi untuk membuat beberapa objek.

3.3.2 Instansiasi Class

Untuk membuat sebuah objek atau sebuah *instance* pada sebuah kelas. Kita menggunakan operator baru. Sebagai contoh, jika anda ingin membuat *instance* dari kelas *string*, kita menggunakan kode berikut :

String str2 = new *String*(“Hello world!”);

or also equivalent to,

String str2 = "Hello";

Figure 7: Classs Instantiation

3.3.3 Variabel Class dan Method

Sebagai tambahan pada contoh *variable*, hal ini juga memungkinkan untuk mendefinisikan *variable* kelas, dimana *variable* milik dari seluruh kelas. Ini berarti bahwa memiliki nilai yang sama untuk semua objek pada kelas yang sama. Mereka juga disebut *static member variables.*

3.4 Method

3.4.1 Apakah Method itu dan mengapa menggunakan Method?

Pada contoh yang telah kita diskusikan sebelumnya, kita hanya memiliki satu *method*, dan itu adalah *main() method*. Didalam *Java*, kita dapat mendefinisikan beberapa *method* yang akan kita panggil dari *method* yang berbeda.

Sebuah *method* adalah bagian terpisah dari kode yang akan dipanggil oleh program utama dan beberapa *method* lainnya untuk menunjukkan beberapa fungsi spesifik.

Berikut adalah karakteristik dari *method* :

1. dapat mengembalikan satu atau tidak ada nilai
2. dia mungkin dapat diterima sebagai beberapa parameter yang dibutuhkan atau tidak ada arameter sama sekali. Parameter juga disebut sebagai fungsi *argument*
3. setelah *method* telah selesai dieksekusi, dia akan kembali pada *method* yang memanggilnya.

Sekarang mengapa kita butuh untuk membuat *method*? Mengapa kita tidak meletakkan semua kode pada sebuah *method* yang sangat besar? Pemecahan masalah disini alah dekomposisi. Kita juga dapat melakukan ini di *Java* dengan mmbuat *method* untuk mengatasi bagian spesifik dari masalah. Mengambil sebuah permasalahan dan memecahkannya menjadi bagian kecil, bagian dapat diatur adalah penting untuk menulis program yang besar.

3.4.2 Memanggil Instance dari Method dan Passing Variabel

Sekarang kita ilustrasikan bagaimana memanggil *method*, mari kita menggunakan kelas *string* sebagai contoh. Anda dapat menggunakan *the Java API documentation* untuk melihat semua *method* dalam kelas *string* yang tersedia. Selanjutnya, kita akan membuat *method* kita sendiri. Tapi sekarang mari kita menggunakan apa yang tersedia.

Untuk memanggil sebuah *instance method,* kita menuliskan :

```
nameOfObject.nameOfMethod( parameters );
```

mari kita mengambil dua contoh yang ditemukan dalam kelas *String*.

Method declaration	**Definition**
public char charAt(int index)	Mengambil karakter pada index.
public boolean equalsIgnoreCase (*String* another*String*)	Membandingkan antar *String*, tidak case sensitive.

Table 3: Method dari Class String

Menggunakan *method* :

```
String str1 = "Hello";
char            x = str2.charAt(0); //will return the character H
                          //simpan pada variabel x

String          str2 = "hello";

//return boolean
boolean  result = str1.equalsIgnoreCase( str1 );
```

3.4.3 Passing Variabel Dalam Method

Pada contoh kita, kita telah mecoba melewati *variable* pada *method*. Bagaimanapun juga kita tidak dapat membedakan antara perbedaan tipe variabel *passing* dalam *Java*. Ada dua tipe data *passing* pada *method*, yang pertama adalah *pass-by-value* dan yang kedua adalah *pass-by-reference*.

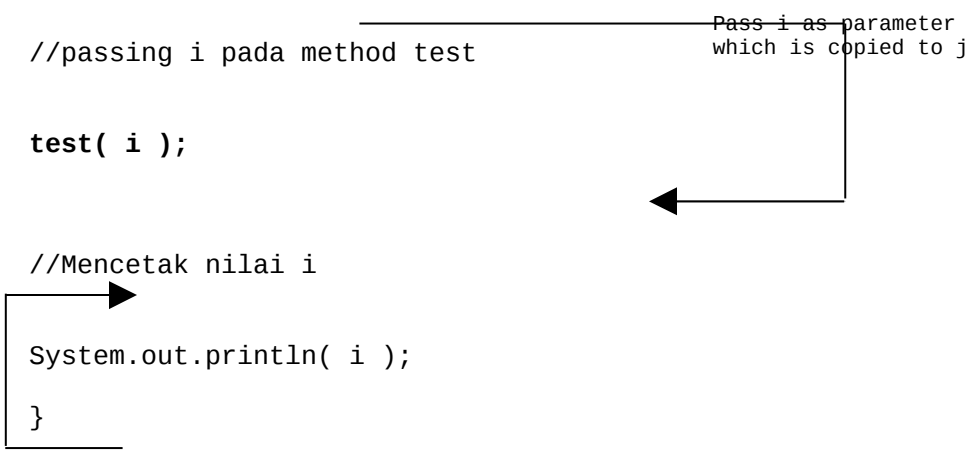
3.4.3.1 Pass-by-Value

Ketika *pass-by-values* terjadi, *method* menggunakan sebuah *copy* pada nilai pada *variable* yang dilewatkan pada *method*. *method* tidak dapat secara langsung dimodifikasi secara *argument* langsung meskipun jika dimodifikasi parameternya selama perhitungan berlangsung.

Contoh :

```
public class TestPassByValue
{

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

```
int i = 10;

//mencetak nilai i

System.out.println( i );


//memanggil method test

//passing i pada method test

test( i );


//Mencetak nilai i

System.out.println( i );

}

public static void test( int j ){

//merubah nilai parameter j

j = 33;

}
}
```



Pada contoh diatas yang telah diberikan, kita memanggil *method tes* dan melewatkan nilai i sebagai *parameter*. Nilai pada i dikopikan pada *variable* pada *method* j. sejak j adalah *variable* pengganti pada *method* tes, dia tidak akan berdampak pada nilai *variable* jika i pada *main* semenjak memiliki perbedaan kopy pada *variable*.

Secara *default*, semua tipe *data primitive* ketika dilewatkan pada sebuah *method* adalah *pass-by-values*

3.4.3.2 Pass-by-reference

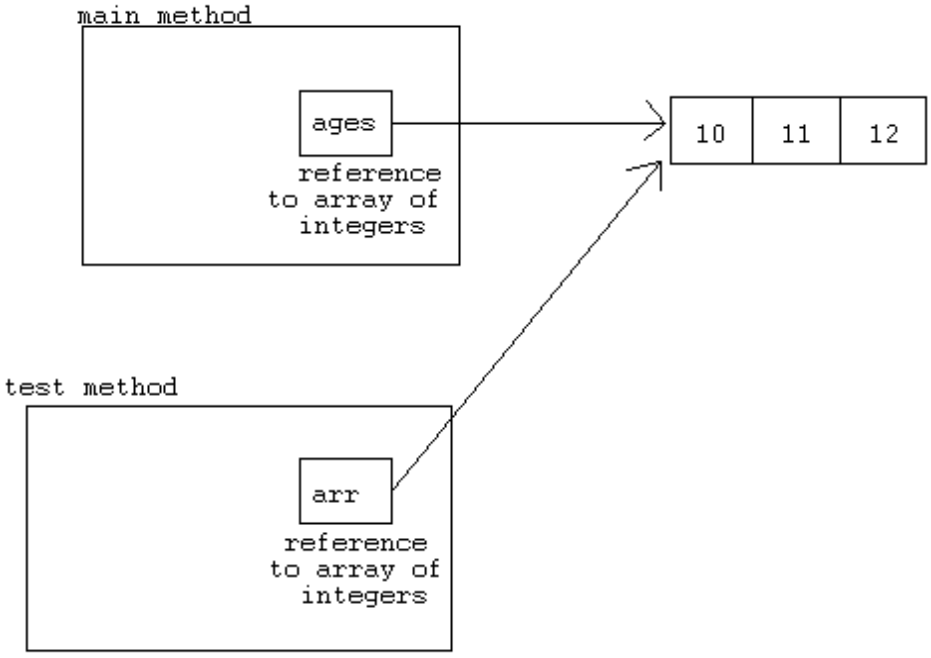
Ketika sebuah *pass-by-reference* terjadi, referensi pada sebuah objek dilewatkan dengan cara memanggil *method*. Hal ini berarti bahwa *method* mengkopi referensi pada *variable* yang dilewatkan pada *method*. Bagaimanapun juga, tidak seperti pada *pass-by-value*, *method* dapat membuat objek actual yang menerangkan *pointing to, since,* meskipun

berbeda keterangan yang digunakan dalam *method*, lokasi dari data yang mereka tunjukkan adalah sama.

contoh :

```
class TestPassByReference
{
	public static void main( String[] args ){
		//membuat array integer
		int []ages	= {10, 11, 12};

		//mencetak nilai array
		for( int i=0; i<ages.length; i++ ){
			System.out.println( ages[i] );
		}

		test( ages );

		for( int i=0; i<ages.length; i++ ){
			System.out.println( ages[i] );
		}
	}

	public static void test( int[] arr ){
		//merubah nilai array
		for( int i=0; i<arr.length; i++ ){
			arr[i] = i + 50;
		}
	}
}
```

Pass ages as parameter which is copied to variable arr



Gambar 2 : Contoh Pass By Reference

Petunjuk Penulisan Program :

Keadaan yang salah tentang nilai oleh referensi di java adalah ketika membuat method swap menggunakan referensi Java, mencatat tentang manipulasi object Java 'by reference' tetapi nilai objeck dari referensi dari method 'by value,'" adalah hasil, anda tidak dapat menulis standart swap method ke swap objek.

3.4.4 Memanggil Method Static

method Static adalah cara yang dapat dipakai tanpa inisialisasi suatu *class* (maksudnya tanpa menggunakan kata kunci yang baru), *method static* mempunyai *class* yang lengkap dan contoh yang tidak pasti (atau objek) dari suatu *class. method static* dibedakan dari contoh *method* di dalam suatu *class* oleh kata kunci *static*.

Untuk memanggil *method static*, ketik,

```
Classname.staticMethodName(params);
```

Contoh dari *static method* yang digunakan :

```
//mencetak data pada layar
System.out.println(“Hello world”);

//convert string menjadi integer
int i = Integer.parseInt(“10”);



String hexEquivalent = Integer.toHexString( 10 );
```

3.4.5 Lingkup Variabel

Sebagai tambahan dari suatu *variable* nama dan tipe data, suatu *variable* mempunyai jangkauan, Jangkauan menentukan dimana program dapat mengakses *variable*, jangkauan juga menentukan kehidupan dari suatu *variable* atau berapa lama *variable* itu berada dalam *memory.* Jangkauan ditentukan oleh dimana deklarasi *variable* di tempatkan di dalam program.

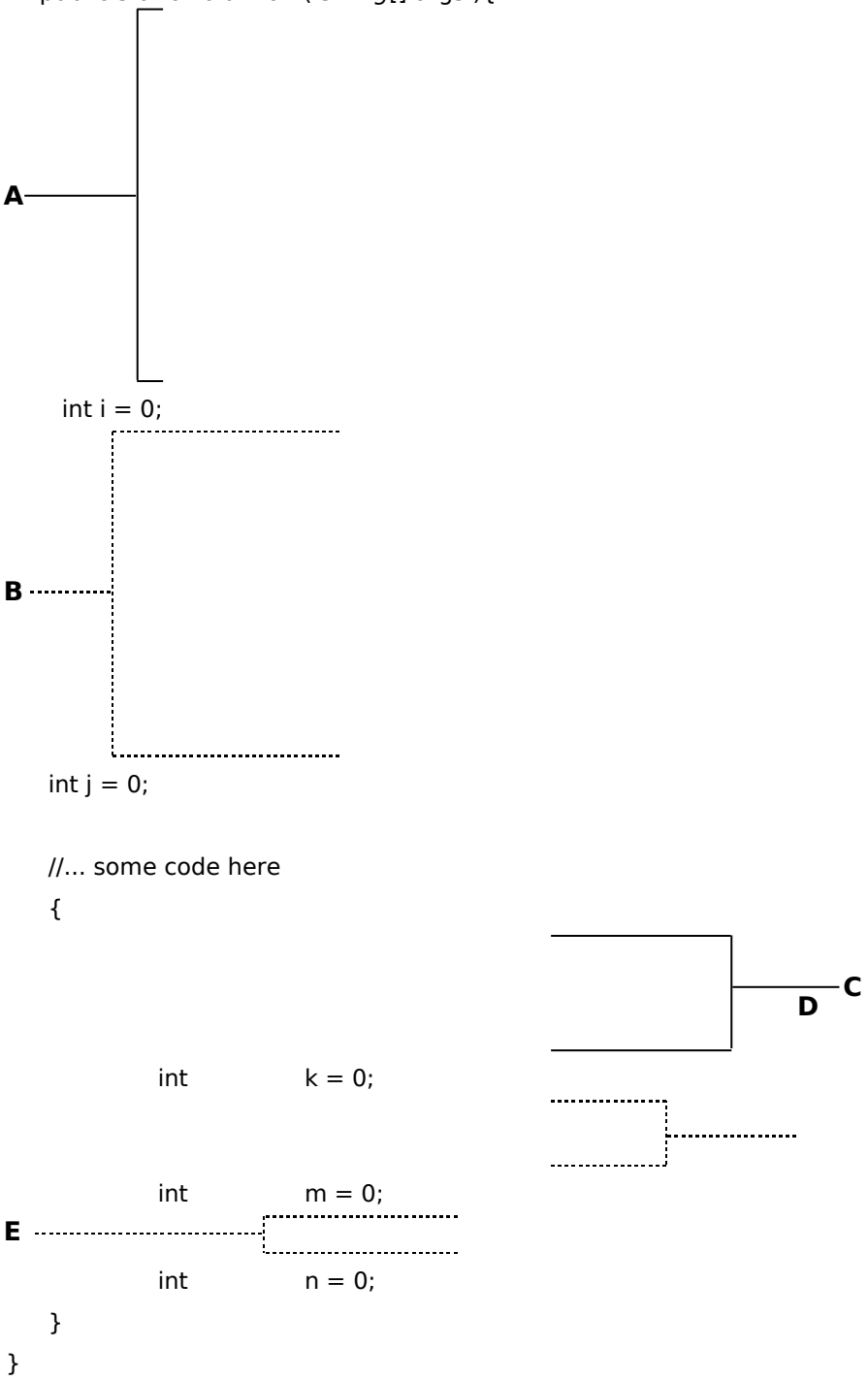
Untuk menyederhanakannya, coba berpikir tentang jangkauan apapun antara kurung kurawal {.....}, diluar kurung kurawal disebut dengan blok terluar, dan didalam kurung kurawal disebut dengan blok terdalam.

Jika kamu mendeklarasikan *variable* di blok luar. Mereka akan terlihat (yaitu, dapat dipakai) Oleh blok bagian dalam, bagaimana pun, jika kamu mendeklarasikan *variable* di blok dalam, kamu tidak bisa harapkan blok terluar untuk melihat itu.

Suatu jangkauan *variable* di dalam blok dimana jika sudah di deklarasi, dimulai dari titik dimana *variable* itu di dklarasikan, dan di blok bagian dalam.

Contoh, yang diberi *code snippet*,

```
public class ScopeExample
{
    public static void main( String[] args ){
      int i = 0;

      int j = 0;

      //... some code here
      {
                int       k = 0;

                int       m = 0;

                int       n = 0;
      }
}
```

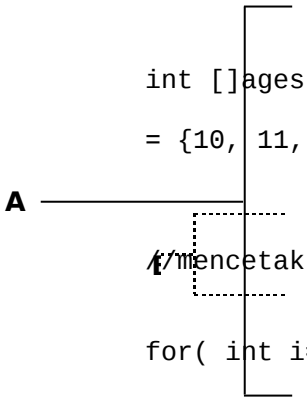


Kode yang kita miliki disini mempunyai lima jangkauan yang ditandai oleh baris dan keterangan yang mewakili jangkauan itu, dengan *variable* i,j,k,m dan n, dan 5 jangkauan A,B,C,D dan E, kita mempunyai beberapa jangkauan *variable* berikut:

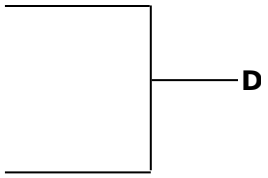
Jangkauan *variable* i adalah A.
Jangkauan *variable* j adalah B.
Jangkauan *variable* k adalah C.
Jangkauan *variable* m adalah D.
Jangkauan *variable* n adalah E.

Sekarang, memberi kedua *method* utama dan menguji di contoh kita sebelumnya,

```
class TestPassByReference
{

public static void main( String[] args ){


//membuat array integer


int []ages

= {10, 11, 12};


//mencetak nilai array


for( int i=0; i<ages.length; i++ ){


System.out.println( ages[i] );


}


test( ages );


//mencetak kembali nilai array


for( int i=0; i<ages.length; i++ ){


System.out.println( ages[i] );


}

}
```



```
public static void test( int[] arr ){

//merubah nilai pada array
     
for( int i=0; i<arr.length; i++ ){


arr[i] = i + 50;


}

}
}
```



Pada *method* pertama, Jangkauan *variables* adalah,

```
ages[ ]   - scope A
i in B    - scope B
i in C    – scope C
```

Pada *method* ujian, Jangkauan *variables* adalah,

```
arr[ ]    - scope D
i in E    - scope E
```

manakala *variable* di deklarasikan,hanya satu *variable* yang di identifikasi atau nama dapat di identifikasi di jangkauan, maksudnya jika kamu mempunyai deklarasi berikut,

```
{
       int test = 10;
       int test = 20;
}
```

Compilermu akan menghasilkan *error* karena kamu perlu mempunyai nama yang lain dari *variable* di satu blok, bagaimanapun, kamu dapat mempunyai dua *variable* dengan nama yang sama, jika mereka tidak dideklarasikan pada blok yang sama, Contoh

```
int test = 0;
System.out.print( test );
//..some code here
{
       int test = 20;
       System.out.print( test );
}
```

Manakala system pertama *out.print* itu memanggil, dia mencetak nilai dari *variable* ujian pertama sejak terdapat pada *variable* jangkauan itu. Yang kedua, *system.out print*, nilai 20 dicetak sejak tertutup ujian jangkauan *variable* itu.

Petunjuk Penulisan program :

Hindari pemberian nama yang sama kepada variabel supaya Anda tidak kebingungan.

3.5 Casting, Converting dan Comparing Objects

Pada bagian ini, kita akan belajar bagaimana mengunakan *typecasting.Typecasting* atau *casting* adalah proses konversi data dari tipe data tertentu ke tipe data yang lain. Kita juga akan belajar bagaimana meng-konversi tipe data *primitive* ke *object* dan sebaliknya. Kemudian, pada akhirnya kita akan belajar bagaimana membandingkan sebuah *object*.

3.5.1 Casting Tipe Primitiv

Casting antara tipe *primitive* mendukung Anda untuk mengkonversikan sebuah *value* dari sebuah tipe data tertentu kepada tipe *primitive* yang lain. Hal ini biasanya terjadi diantara tipe data numerik.

Ada sebuah tipe data *primitive* yang tetap tidak dapat kita *casting*, dan dia adalah tipe data *boolean*.

Sebagai contoh dari *typecasting* adalah pada saat Anda menyimpan sebuah *integer* kepada sebuah variabel dengan tipe data *double*. Sebagai contoh:

```
int   numInt = 10;
double  numDouble = numInt; //implicit cast
```

Pada contoh ini dapat kita lihat bahwa, walaupu variabel yang dituju *(double)* memiliki nilai yang lebih besar daripada nilai yang akan kita tempatkan didalamnya, data tersebut secara implisit dapat kita *casting* ke tipe data *double*.

Contoh yang lain adalah apabila kita ingin untuk melakukan *typecasting* sebuah int ke *char* atau sebaliknya. Sebuah karakter akan dapat digunakan sebagai int karena setiap karakter memiliki sebuah nilai numerik yang merepresentasikan posisinya dalam satu *set* karakter. Jika sebuah *variable* memiliki nilai 65, maka *cast* (char) i akan menghasilkan nilai 'A'. Numerik kode yang merepresentasikan kapital A adalah 65, berdasarkan karakter *set* ASCII, dan *Java* telah mengadopsi bagian ini untuk mendukung karakter.

```
char valChar = 'A';
int  valInt = valChar;
System.out.print( valInt ); //casting explisit: keluaran 65
```

Ketika kita men-*convert* data yang bertipe besar ke tipe data yang lebih kecil, kita harus menggunakan ***explicit cast***. *Explicit casts* mengikuti bentuk sebagai berikut :

(dataType)value

dimana,

dataType, adalah nama dari tipe data yang Anda *convert*
value, adalah pernyataan yang dihasilkan pada nilai dari *the source type*.

Sebagai contoh,

```
double  valDouble = 10.12;
int     valInt = (int)valDouble; //men-convert valDouble ke tipe int

double  x = 10.2;
int     y = 2;

int     result = (int)(x/y); //hasil typecast operasi ke int
```

3.5.2 Casting Objects

Instances dari *class-class* juga dapat di pilih ke *instance-instance* dari *class-class* yang lain, dengan **satu batasan**: ***class-class* sumber dan tujuan harus terhubung dengan mekanisme *inheritance*; satu *class* harus menjadi sebuah subclass terhadap *class* yang lain.** kita akan akan menjelaskan mengenai inheritance pada kesempatan selanjutnya.

Sejalan dengan pemilihan nilai *primitive* untuk tipe yang lebih besar, beberapa *object* mungkin tidak membutuhkan untuk dipilih secara *explisit*. Faktanya, karena sebuah semua *subclass* terdiri atas informasi yang sama, Anda dapat menggunakan *instance* dari *subclass* diamanpun sebuah *superclass* diharapkan berada.

Sebagai contoh, mempertimbangkan *methode* yang memiliki dua *argument*, satu tipe *object* dan tipe *window* yang lain. Anda dapat melewatkan *instance* dari beberapa *class* untuk *argument object* karena semua *class java* adalah *subclass* dari *object*. Untuk *argument window*, anda dapat melewatkannya kedalam *subclassnya,* seperti *dialog*, *FileDialog*, dan *frame*. Ini benar dimanapun dalam program, bukan hanya dalam memanggil *methode*. Jika anda mempunyai variabel yang didefinisikan sebagai *window class*, anda dapat memberikan *object* dari kelas tersebut atau dari *subclassnya* untuk variabelnya tanpa pemilihan.

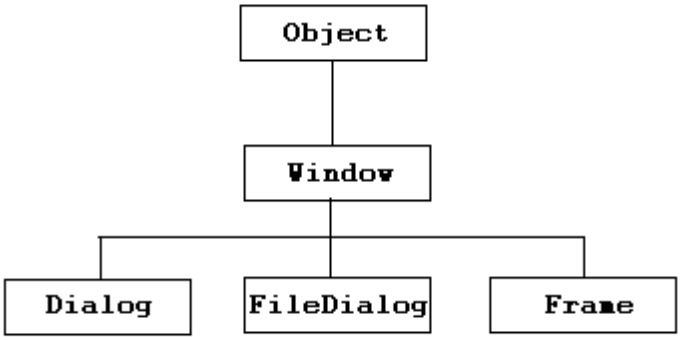


Figure 8: Contoh Hierarchy Class

Ini dibenarkan dalam kasus yang berkebalikan, dan Anda dapat menggunakan *superclass* ketika sebuah *subclass* dibentuk. Ada yang didapatkan dalam kasus ini, bagaimanapun: **Karena *subclass* terdiri dari lebih banyak kemungkinan aksi daripada *superclassnya*, terdapat kehilangan dalam keseimbangan keterlibatan.** *Object superclass* itu mungkin tidak memiliki semua kemungkinan aksi yang diperlukan untuk aksi pada tempat dari *object subclass* berada. Sebagai contoh jika anda memiliki operasi yang memanggil *methode* dalam *object* dari *class integer*, mengunakan *object* dari *class Number* tidak akan terdiri dari benyak *methode* yang dispesifikasikan dalam *integer. error* terjadi jika Anda mencoba untuk memanggil *methode* yang tidak memiliki *object* tujuan.

Untuk mengunakan *object-object superclass* dimana *object-object subclass* diharapkan, anda harus memilih mereka secara eksplisit. Anda tidak akan kehilangan beberapa informasi dalam pemilihan, tapi anda memperoleh keuntungan dari semua *method* dan variabel yang mendefinisikan *subclass*. Untuk memilih sebuah *object* ke *class* yang lain, Anda menggunakan operasi yang sama sebagaimana untuk tipe-tipe *primitive* :

Untuk memilih,

(classname)object

dimana,

classname, adalah nama dari *class* tujuan.

object, adalah sesuatu yang mengarah pada sumber *object*.

- **Catatan:** pemilihan ini membuat referensi ke *object* yang lama dari tipe namaclass; *object* yang lama melanjutkan aksi seperti yang telah ada sebelumnya.

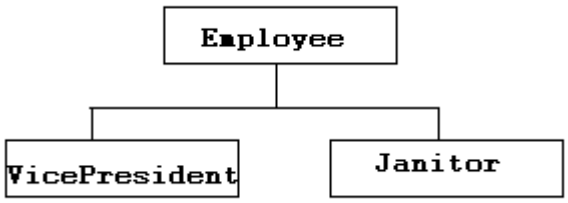


Figure 9: Class Hierarchy untuk superclass Employee

Contoh berikut memilih sebuah *instance* dari *class VicePresident* ke sabuah *instance* dari *class Employee*; *VicePresident* adalah sebuah dari *Employee* dengan lebih banyak information, dimana disini mendefinisikan bahwa *VicePresident* memilihi *executive washroom privileges*,

```
Employee emp = new Employee();
VicePresident veep = new VicePresident();
emp = veep; // tidak adah pemilihan yang diperlukan untuk penggunaan yang
cenderung naik
veep = (VicePresident)emp; // Harus memilih dengan pemilihan secara
eksplisit
```

3.5.3 Convert Tipe Primitive ke Object Dan Sebaliknya

Satu hal yang tidak dapat Anda lakukan pada beberapa keadaan yaitu pemilihan dari sebuah *object* ke sebuah tipe data *primitive*, atau *vice versa*. Tipe-tipe *primitive* dan *object*

adalah sesuatu yang sangat berbeda dalam *Java*, dan Anda tidak bisa secara langsung memilih diantara dua atau saling menukar diantara keduanya.

Sebagai sebuah alternatif, *package* ***java.lang*** yang terdiri atas *class-class* yang sesuai untuk setiap tipe data primitivenya yaitu : *Float, Boolean, Byte*, dan sebagainya. Kebanyakan dari *class-class* ini memiliki nama yang sama seperti tipe datanya, kecuali jika nama *classnya* diawali dengan huruf *capital(Short -> sort, Double -> double* dan sebagainya). Juga dua *class* memiliki nama yang berbeda dari tipe data yang sesuai : *Character* digunakan untuk variabel *char* dan *Integer* untuk variabel *int*. **(Disebut dengan *Wrapper Classes*)**

Java merepresentasikan *type data* dan versi *classnya* dengan sangat berbeda, dan sebuah program tidak akan berhasil tercompile jika Anda menggunakan hanya satu ketika yang lain juga diperlukan.

Menggunakan *class-class* yang sesuai untuk setiap tipe *primitive*, anda dapat membuat sebuah *object* yang memiliki nilai yang sama.

Contoh :

```
//Pernyataan berikut membentuk sebuah instance bertype Integer
// class dengan nilai integer 7801 (primitive -> Object)
Integer dataCount = new Integer(7801);

//Pernyataan berikut meng-converts sebuah object Integer ke
//tipe data primitive int nya. Hasilnya adalah sebuah int //dengan nilai 7801

int newCount = dataCount.intValue();

// Anda perlu suatu translasi biasa pada program
// yang meng-convert sebuah String ke sebuah tipe numeric, //seperti suatu int
// Object->primitive
String pennsylvania = "65000";
int penn = Integer.parseInt(pennsylvania);
```

- **PERHATIAN:** *class Void* tidak mewakili sesuatu dalam *Java*, jadi disini tidak ada alasan menggunakannya ketika melakukan translasi antara nilai *primitive* dan *object*. Ini adalah penjelasan mengenai kata kunci *void*, dimana digunakan dalam definisi *method* untuk mengindikasikan bahwa *methode* tidak memiliki sebuah nilai kembalian.

3.5.4 Comparing Objects

Dalam diskusi kita sebelumnya, kita mempelajari tentang operator untuk membandingkan nilai —sama, tidak sama, lebih kecil daripada, dan sebagainya. Operator ini yang paling banyak bekerja hanya pada tipe *primitive*, bukan pada *object*. Jika Anda berusaha untuk menggunakan nilai lain sebagai *operands*, *Compiler Java* akan menghasilkan *error*.

Pengecualian untuk aturan ini adalah operator untuk persamaan : == (sama) dan != (tidak). Ketika dinampilkan ke *object,* operator ini tidak akan melakukan apa yang sebenarnya anda inginkan. Malahan mengecheck jika satu *object* memilki nilai yang sama seperti *object* lain, mereka mengenali jika kedua sisi dari operator menunjuk *object* yang sama.

Untuk membandingkan *instances* dari sebuah *class* dan memiliki hasil yang berarti, Anda

harus mengimplementasikan *method* khusus dalam *class* Anda dan memanggil *method* tersebut. Sebuah contoh yang baik untuk ini adalah *class String*.

Sangat mungkin memiliki dua *object String* yang memiliki nilai yang sama. Jika Anda menggunakan operator == untuk membandingkan *object* ini, bagaimanapun, kita akan mempertimbangkan nilai yang tidak sama. Walaupun isinya sesuai mereka bukan merupakan *object* yang sama.

Untuk melihat jika dua *object String* memiliki nilai yang sesuai, sebuah *method* dari *class* yang disebuat dengan *equals()* digunakan. *Method* menguji setiap *character* dalam *string* dan mengembalikan nilai *true* jika dua *string* memiliki nilai yang sama.

Kode berikut mengilustrasikan hal tersebut,

```
class EqualsTest {
  public static void main(String[] arguments) {
    String str1, str2;
    str1 = "Free the bound periodicals.";
    str2 = str1;

    System.out.println("String1: " + str1);
    System.out.println("String2: " + str2);
    System.out.println("Same object? " + (str1 == str2));

    str2 = new String(str1);

    System.out.println("String1: " + str1);
    System.out.println("String2: " + str2);
    System.out.println("Same object? " + (str1 == str2));
    System.out.println("Same value? " + str1.equals(str2));
  }
 }
```

Output program ini adalah sebagai berikut ,

OUTPUT:

```
String1: Free the bound periodicals.
String2: Free the bound periodicals.
Same object? true
String1: Free the bound periodicals.
String2: Free the bound periodicals.
Same object? false
Same value? True
```

Sekarang mari mendiskusikan tentang kode.

```
String str1, str2;
str1 = "Free the bound periodicals.";
```

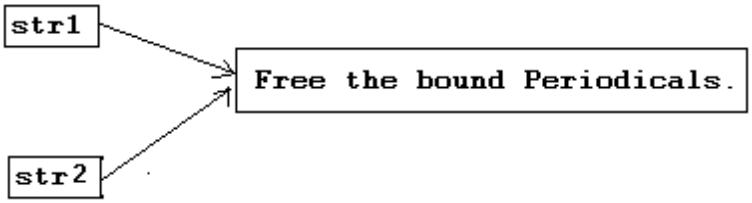


Figure 10: Keduanya mengarah ke object yang sama

Bagian pertama dari program ini mendeklarasikan dua variabel (str1 dan str2), memberikan literal "*Free the bound periodicals.*" untuk str1, dan kemudian memberi nilai tersebut untuk str2. Seperti yang Anda pelajari sebelumnya, str1 dan str2 sekarang menunjuk ke *object* yang sama, dan uji kesamaan membuktikan hal tersebut.

str2 = new *String*(str1);

Padabagian yang kedua dari program ini, anda membuat *object String* baru dengan nilai yang sama sebagai str1 dan memberi str2 ke *object* baru *String* tersebut. Sekarang Anda memiliki dua *object string* yang berbeda yaitu str1 dan str2, keduanya memilki nilai yang sama.*Test* mereka untuk melihat jika meeka *object* yang sama dengan menggunakan *operator* == mengembalikan nilai yang diinginkan : *false*—mereka buka *object* yang sama dalam *memory*. *Test* mereka menggunakan *method equals()* juga mengembalikan jawaban yang diinginkan: *true*—mereka memiliki niali yang sama.

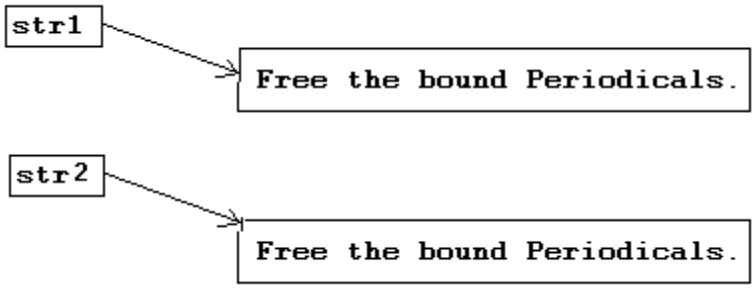


Figure 11: Sekarang mengarah pada object yang berbeda

- **Catatan:** Mengapa Anda tidak dapat hanya menggunakan *literal* yang lain ketika Anda mengubah str2, lebih dari menggunakan *new*? *String literals* diandalkan dalam *Java*; jika Anda membuat sebuah *string* menggunakan *literal* dan kemudian menggunakan *literal* yang lain dengan *character* yang sama, *Java* cukup mengetahui untuk memberikan Anda *object String* yang pertama kembali. kedua *String* adalah *object* yang sama; Anda harus menghidari langkah anda untuk membuat dua *object* terpisah.

3.5.5 Menentukan Class dari sebuah Object

Ingin menemukan apakah sebuah *class object* itu? Disini langkah untuk melakukannya untuk sebuah *object* yang diberikan sebagai kunci variabel :

1. ***Method getClass()*** mengembalikan sebuah *object Class* (dimana *Class* itu sendiri merupakan sebuah *class*) yang memilki sebuah *method* yang disebut *getName()*. Pada bagiannya, *getName()* mengembalikan sebuah *string* yang mewakili nama *class*.

Sebagai contoh,

```
String name = key.getClass().getName();
```

2. operator InstanceOf

instanceOf memiliki dua *operands*: suatu mengarahke sebuah *object* pada sebelah kiri dan nama *class* pada sebelah kanan. pernyataan mengembalikan nilai *true* atau *false* tergantung pada apakah *object* adalah sebuah *instance* dari penamaan *class* atau beberapa dari *subclass* milik *class* tersebut.

Sebagai contoh,

```
boolean ex1 = "Texas" instanceof String; // true
Object pt = new Point(10, 10);
boolean ex2 = pt instanceof String; // false
```

3.6 Latihan

3.6.1 Mendefinisikan Istilah

Dengan kata-kata Anda sendiri, definisikan istilah-istilah berikut ini :

1. *Class*
2. *Object*
3. *Instantiate*
4. *Instance Variable*
5. *Instance Method*
6. *Class Variables* atau *static member variables*
7. *Constructor*

3.6.2 Java Scavanger Hunt

Pipoy adalah suatu anggota baru dalam bahasa pemrograman *Java*. Dia hanya memperdengarkan bahwa telah ada APIs siap pakai dalam *Java* yang salah satunya dapat digunakan dalam program mereka, dan ia ingin sekali untuk mengusahakan mereka keluar.

Masalahnya adalah, Pipoy tidak memiliki *copy* dari dokumentasi *Java*, dan dia juga tidak memiliki *acces internet*, jadi tidak ada jalan untuknya untuk menunjukkan *Java APIs*.

Tugas Anda adalah untuk membantu Pipoy memperhatikan APIs *(Application Programming Interface)*. Anda harus menyebutkan *class* dimana seharusnya *method* berada, deklarasi *method* dan penggunaan contoh yang dinyatakan *method*.
Sebagai contoh, jika Pipoy ingin untuk mengetahui *method* yang mengknversisebuah *String* ke *integer*,jawaban Anda seharusnya menjadi:

Class: Integer
Method Declaration: public static int parseInt(*String* value)
Sample Usage:

```
String      strValue = "100";
int         value = Integer.parseInt( strValue );
```

yakinkan bahwa *snippet* dari kode yang Anda tulis dalam contoh Anda menggunakan *compiles* dan memberi *output* jawaban yang benar, jadi tidak membingungkan Pipoy. **(Hint: Semua *methods* adalah dalam *java.lang package*).** Dalam kasus dimana Anda dapat menemukan lebih banyak *methods* yang dapat menyelesaikan tugas, berikan hanya satu.

Sekarang mari memulai pencarian!

1. Perhatikan sebuah *method* yang diuji jika *String* pasti diakhiri *suffix* yang pasti. Sebagai contoh, jika diberikan *string* "*Hello*", *Method* harus mengembalikan nilai *true suffix* yang diberikan adalah "lo", dan *false* jika *suffix* yang diberikan adalah "alp".
2. Perhatikan untuk *method* yang mengenali *character* yang mewakili sebuah digit yang spesifik dalam *radix* khusus. Sebagai contoh, jika *input* digit adalah 15, dan *the radix* adalah 16, *method* akan mengembalikan *Character* F, sejak F adalah representasi *hexadecimal* untuk angka 15 (berbasis 10).
3. Perhatikan untuk *method* yang mengakhiri *running Java Virtual Machine* yang sedang berjalan
4. Perhatikan untuk *method* yang memperoleh lantai dari sebuah nilai *double*. Sebagai contoh, jika Saya *input* a 3.13, *method* harus mengembalikan nilai 3.
5. Perhatikan untuk *method* yang mengenali jika *character* yang dipakai adalah sebuah digit. Sebagai contoh, jika Saya *input* '3', dia akan mengembalikan nilai *true*.

BAB 4

Membuat *Class* Sendiri

4.1 Tujuan

Setelah kita mempelajari penggunaan *class* dari *Java Class Library*, kita akan mempelajari bagaimana menuliskan sebuah *class* sendiri. Pada bagian ini, untuk mempermudah pemahaman pembuatan *class*, kita akan membuat contoh *class* dimana akan ditambahkan beberapa data dan fungsi – fungsi lain.

Kita akan membuat *class* yang mengandung informasi dari Siswa dan operasi – operasi yang dibutuhkan pada *record* siswa.

Beberapa hal yang perlu diperhatikan pada *syntax* yang digunakan pada bab ini dan bagian lainnya :

*	-	Menandakan bahwa terjadi lebih dari satu kejadian dimana elemen tersebut diimplementasikan
<description>	-	Menandakan bahwa Anda harus memberikan nilai pasti pada bagian ini
[]	-	Indikasi bagian optional

Pada akhir pembahasan, diharapkan siswa dapat :

- Membuat kelas mereka sendiri
- Mendeklarasikan atribut dan *method* pada *class*
- Menggunakan referensi *this* untuk mengakses *instance data*
- Membuat dan memanggil *overloaded method*
- Mengimport dan membuat *package*
- Menggunakan *access modifiers* untuk mengendalikan akses terhadap *class member*

4.2 Mendefinisikan *Class* Anda

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

Dalam pendefinisian *class*, dituliskan :

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

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

Petunjuk Penulisan Program :

Perhatikan bahwa pada class teratas, access modifier yang diperbolehkan adalah public dan package (bila tidak terdapat penulisan keyword access modifier pada kelas)

Pada bagian ini, kita akan membuat sebuah *class* yang berisi *record* dari siswa. Jika kita telah mengidentifikasikan tujuan dari pembuatan kelas, maka dapat dilakukan pemberian nama yang sesuai. Nama yang tepat pada *class* ini adalah *StudentRecord*.

Untuk mendefinisikan *class*, kita tuliskan :

```
public class StudentRecord
{
       //area penulisan kode selanjutnya
}
```

dimana,

Public	-	*Class* ini dapat diakses dari luar *package*
Class	-	*Keyword* yang digunakan di pembuatan *class* Java
StudentRecord	-	Identifier yang menjelaskan *class*

Petunjuk Penulisan Program :

1. Pertimbangkan nama yang tepat untuk class. Jangan gunakan nama acak dan singkat seperti XYZ.
2. Nama class harus dimulai dengan huruf kapital
3. Nama file dari class harus sama dengan nama public class

4.3 Deklarasi Atribut

Dalam pendeklarasian atribut, kita tuliskan :

```
<modifier> <type> <name> [= <default_value>];
```

Langkah selanjutnya adalah mengurutkan atribut yang akan diisikan pada *class.* Untuk setiap informasi, urutkan juga tipe data yang yang tepat untuk digunakan. Contohnya, Anda tidak mungkin menginginkan untuk menggunakan tipe data *integer* untuk nama siswa, atau tipe data *string* pada nilai siswa.

Berikut ini adalah contoh informasi yang akan diisikan pada *class StudentRecord* :

```
name            - String
address         - String
age             - Int
math grade      - double
english grade   - double
science grade   - double
average grade   - double
```

Anda dapat menambahkan informasi lain jika diperlukan.

4.3.1 Instance Variable

Jika kita telah menuliskan seluruh atribut yang akan diisikan pada *class*, selanjutnya kita akan menuliskannya pada kode. Jika kita menginginkan bahwa atribut – atribut tersebut adalah unik untuk setiap *object* (dalam hal ini untuk setiap siswa), maka kita harus mendeklarasikannya sebagai *instance variable :*

Sebagai contoh :

```
public class StudentRecord
{
	private String name;
	private String address;
	private int age;
	private double mathGrade;
	private double englishGrade;
	private double scienceGrade;
	private double average;

	//area penulisan kode selanjutnya
}
```

dimana,

private disini menjelaskan bahwa variabel tersebut hanya dapat diakses oleh *class* itu sendiri. *Object* lain tidak dapat menggunakan variabel tersebut secara langsung. Kita akan membahas tentang kemampuan akses pada pembahasan selanjutnya.

Petunjuk Penulisan Program :

1. Deklarasikan seluruh instance variable pada awal penulisan class
2. Deklarasikan variable per baris
3. Penulisan instance variable, termasuk juga variabel lain harus dimulai dengan huruf kecil
4. Gunakan tipe data yang tepat pada setiap variabel
5. Deklarasikan instance variable sebagai private supaya hanya method pada class itu sendiri yang dapat mengaksesnya.

4.3.2 Class Variable atau Static Variables

Disamping *instance variable*, kita juga dapat mendeklarasikan *class variable* atau variabel yang dimiliki *class* sepenuhnya. Nilai pada variabel ini sama pada semua *object* di *class* yang sama. Anggaplah kita menginginkan jumlah dari siswa yang dimiliki dari seluruh kelas, kita dapat mendeklarasiakan satu *static variable* yang akan menampung nilai tersebut. Kita beri nama variabel tersebut dengan nama *studentCount*.

Berikut penulisan *static variable :*

```
public class StudentRecord
{
	//area deklarasi instance variables

	private static int studentCount;

	//area penulisan kode selanjutnya
}
```

Kita gunakan *keyword : 'static'* untuk mendeklarasikan bahwa variabel tersebut adalah *static*.

Maka keseluruhan kode yang dibuat terlihat sebagai berikut :

```
public class StudentRecord
{
	private String name;
	private String address;
	private                          int

	age;
	private double mathGrade;
	private double englishGrade;
	private double scienceGrade;
	private double average;

	private static int studentCount;

	//area penulisan kode selanjutnya
}
```

4.4. Deklarasi *Methods*

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

Dalam pendeklarasian *method*, kita tuliskan :

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

dimana,

<modifier> dapat menggunakan beberapa *modifier* yang berbeda
<returnType> dapat berupa seluruh tipe data, termasuk *void*
<name> *identifier* atas *class*
<parameter> ::= <tipe_parameter> <nama_parameter>[,]

4.4.1 Accessor Methods

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

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

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

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

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

dimana,

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

Pernyataan berikut,

```
return name;
```

dalam program kita menandakan akan ada pengembalian nilai dari *instance variable name* pada pemanggilan *method*. Perhatikan bahwa *return type* dari *method* harus sama dengan tipe data terhadap data pada pernyataan *return*. Anda akan mendapatkan pesan kesalahan sebagai berikut bila tipe data yang digunakan tidak sama :

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

Contoh lain dari penggunaan *accessor method* adalah ***getAverage***,

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

              return result;
       }
}
```

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

4.4.2 Mutator Methods

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

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

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

dimana,

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

Pernyataan berikut :

```
name = temp;
```

mengidentifikasi nilai dari temp sama dengan name dan mengubah data pada *instance variable name*.

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

4.4.3 Multiple Return Statements

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

Sebagai contoh, perhatikan *method* berikut ini :

```
public String getNumberInWords( int num ){

	String defaultNum = "zero";

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

4.4.4 Static Methods

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

```
public class StudentRecord
{
	private static int studentCount;

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

dimana,

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

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

Petunjuk Penulisan Program :

1. Nama method harus dimulai dengan huruf kecil
2. Nama method harus berupa kata kerja
3. Gunakan dokumentasi sebelum mendeklarasikan sebuah method. Anda dapat Menggunakan JavaDoc.

4.4.5 Contoh Kode Program dari class StudentRecord

Berikut ini adalah kode untuk *class* StudentRecord :

```
public class StudentRecord
{
	private String name;
	private String address;
	private int age;
	private double mathGrade;
	private double englishGrade;
	private double scienceGrade;
	private double average;

	private static int studentCount;
```

```
	/**
	* Menghasilkan nama dari Siswa
	*/
	public String getName(){
		return name;
	}

	/**
	* Mengubah nama siswa
	*/
	public void setName( String temp ){
		name = temp;
	}

	// area penulisan kode lain
	/**
	* Menghitung rata – rata nilai Matematik, Bahasa Inggris, * * Ilmu
	Pasti
	*/
	public double getAverage(){
		double result = 0;
		result = ( mathGrade+englishGrade+scienceGrade )/3;

		return result;
	}

	/**
	* Menghasilkan jumlah instance StudentRecord
	*/
	public static int getStudentCount(){
		return studentCount;
	}
}
```

Berikut ini contoh kode dari *class* yang mengimplementasikan *class* StudentRecord :

```
public class StudentRecordExample
{
	public static void main( String[] args ){

		//membuat 3 object StudentRecord
		StudentRecord annaRecord = new StudentRecord();
		StudentRecord beahRecord = new StudentRecord();
		StudentRecord crisRecord = new StudentRecord();

		//Memberi nama siswa
		annaRecord.setName("Anna");
		beahRecord.setName("Beah");
```

```
                crisRecord.setName("Cris");

                //Menampilkan nama siswa “Anna”
                System.out.println( annaRecord.getName() );

                //Menampilkan jumlah siswa
                System.out.println("Count="+StudentRecord.getStudentCount()
                );
        }
}
```

Output dari program adalah sebagai berikut :

```
Anna
Student Count = 0
```

4.5. Referensi *this*

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

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

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

```
age = age;
```

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

```
this.<namaInstanceVariable>
```

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

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

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

CATATAN : Anda hanya dapat menggunakan referensi tipe ini terhadap *instance variable* dan BUKAN *static* ataupun class variabel.

4.6. *Overloading Methods*

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

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

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

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

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

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

```
        System.out.println("English Grade:" + eGrade);
        System.out.println("Science Grade:" + sGrade);
}
```

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

```
public static void main( String[] args )
{
        StudentRecord annaRecord = new StudentRecord();

        annaRecord.setName("Anna");
        annaRecord.setAddress("Philippines");
        annaRecord.setAge(15);
        annaRecord.setMathGrade(80);
        annaRecord.setEnglishGrade(95.5);
        annaRecord.setScienceGrade(100);

        //overloaded methods
        annaRecord.print(

        annaRecord.getName() );
        annaRecord.print(

        annaRecord.getEnglishGrade(),
                        annaRecord.getMathGrade(),
                        annaRecord.getScienceGrade());
}
```

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

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

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

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

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

1. Nama yang sama
2. Parameter yang berbeda
3. Nilai kembalian (*return*) bisa sama ataupun berbeda

4.7. Deklarasi Constructor

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

Berikut ini adalah *property* dari *Constructor* :

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

Untuk mendeklarasikan *constructor*, kita tulis,

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

4.7.1 Default Constructor

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

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

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

4.7.2 Overloading Constructor

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

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

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

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


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

4.7.3 Menggunakan Constructor

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

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

              //area penulisan kode selanjtunya
       }
```

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

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

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

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


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

4.7.4 Pemanggilan Constructor Dengan this()

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

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

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

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

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

4.8. *Packages*

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

4.8.1 *Mengimport Packages*

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

Penulisan import *package* dapat dilakukan seperti dibawah ini :

```
import <namaPaket>;
```

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

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

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

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

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

4.8.2 Membuat Package

Untuk membuat *package*, dapat dilakukan dengan menuliskan :

```
package <packageName>;
```

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

Langkah pertama yang harus dilakukan adalah membuat folder dengan nama schoolClasses. Salin seluruh *class* yang ingin diletakkan pada *package* dalam folder ini. Kemudian tambahkan kode deklarasi *package* pada awal file. Sebagai contoh :

```
package schoolClasses;

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

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

4.8.3 Pengaturan CLASSPATH

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

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

Asumsikan kita mengkompilasi dan menjalankan *class* StudentRecord :

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

C:\schoolClasses>java StudentRecord
Exception in thread "main" java.lang.NoClassDefFoundError: StudentRecord
(wrong name: schoolClasses/StudentRecord)
at java.lang.ClassLoader.defineClass1(Native Method)
at java.lang.ClassLoader.defineClass(Unknown Source)
at java.security.SecureClassLoader.defineClass(Unknown Source)
at java.net.URLClassLoader.defineClass(Unknown Source)
at java.net.URLClassLoader.access$100(Unknown Source)
at java.net.URLClassLoader$1.run(Unknown Source)
at java.security.AccessController.doPrivileged(Native Method)
at java.net.URLClassLoader.findClass(Unknown Source)
at java.lang.ClassLoader.loadClass(Unknown Source)
at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
at java.lang.ClassLoader.loadClass(Unknown Source)
at java.lang.ClassLoader.loadClassInternal(Unknown Source)
```

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

Pengaturan classpath pada *Windows* dilakukan pada *command prompt* :

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

dimana C:\ adalah direktori dimana kita menempatkan *package*. Setelah mengatur classpath, kita dapat menjalankan program di mana saja dengan mengetikkan :

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

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

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

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

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

dan untuk sistem UNIX :

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

4.9. *Access Modifiers*

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

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

4.9.1 Akses Default (Package Accessibility)

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

```
public class StudentRecord
{
	//akses dasar terhadap variabel
	int name;

	//akses dasar terhadap metode
	String getName(){
		return name;
	}
}
```

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

4.9.2 Akses Public

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

```
public class StudentRecord
{
	//akses dasar terhadap variabel
	public int name;

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

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

4.9.3 Akses Protected

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

```
public class StudentRecord
{
	//akses pada variabel
	protected int name;

	//akses pada metode
	protected String getName(){
```

```
                return name;
        }
}
```

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

4.9.4 Akses Private

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

```
public class StudentRecord
{
       //akses dasar terhadap variabel
       private int name;

       //akses dasar terhadap metode
       private String getName(){
              return name;
       }
}
```

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

Petunjuk Penulisan Program :

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

4.10. Latihan

4.10.1 Entry Buku Alamat

Tugas Anda adalah membuat sebuah *class* yang memuat data-data pada buku alamat. Tabel berikut mendefinisikan informasi yang dimiliki oleh buku alamat.

Attribut	Deskripsi
Nama	Nama Lengkap perseorangan
Alamat	Alamat Lengkap
Nomor Telepon	Nomor telepon personal
Alamat E-Mail	Alamat E-Mail personal

Tabel 1: Atribut dan Deskripsi Atribut

Buat implementasi dari *method* sebagai berikut :

1. Menyediakan *accessor* dan *mutator method* terhadap seluruh atribut
2. Constructor

4.10.2 Buku Alamat

Buat sebuah *class* buku alamat yang dapat menampung 100 data. Gunakan *class* yang telah dibuat pada nomor pertama. Anda harus mengimplementasikan *method* berikut pada buku alamat :

1. Memasukkan data
2. Menghapus data
3. Menampilkan seluruh data
4. Update data

Bab 5
Algoritma *Sorting*

5.1 Tujuan

Sorting adalah proses menyusun elemen – elemen dengan tata urut tertentu dan proses tersebut terimplementasi dalam bermacam aplikasi. Kita ambil contoh pada aplikasi perbankan. Aplikasi tersebut mampu menampilkan daftar *account* yang aktif. Hampir seluruh pengguna pada sistem akan memilih tampilan daftar berurutan secara *ascending* demi kenyamanan dalam penelusuran data.

Beberapa macam algoritma *sorting* telah dibuat karena proses tersebut sangat mendasar dan sering digunakan. Oleh karena itu, pemahaman atas algoritma – algoritma yang ada sangatlah berguna.

Setelah menyelesaikan pembahasan pada bagian ini, anda diharapkan mampu :

1. Memahami dan menjelaskan algoritma dari *insertion sort, selection sort, merge sort* dan *quick sort*.
2. Membuat implementasi pribadi menggunakan algoritma yang ada

5.2 *Insertion Sort*

Salah satu algoritma *sorting* yang paling sederhana adalah *insertion sort*. Ide dari algoritma ini dapat dianalogikan seperti mengurutkan kartu. Penjelasan berikut ini menerangkan bagaimana algoritma *insertion sort* bekerja dalam pengurutan kartu. Anggaplah anda ingin mengurutkan satu set kartu dari kartu yang bernilai paling kecil hingga yang paling besar. Seluruh kartu diletakkan pada meja, sebutlah meja ini sebagai meja pertama, disusun dari kiri ke kanan dan atas ke bawah. Kemudian kita mempunyai meja yang lain, meja kedua, dimana kartu yang diurutkan akan diletakkan. Ambil kartu pertama yang terletak pada pojok kiri atas meja pertama dan letakkan pada meja kedua. Ambil kartu kedua dari meja pertama, bandingkan dengan kartu yang berada pada meja kedua, kemudian letakkan pada urutan yang sesuai setelah perbandingan. Proses tersebut akan berlangsung hingga seluruh kartu pada meja pertama telah diletakkan berurutan pada meja kedua.

Algoritma *insertion sort* pada dasarnya memilah data yang akan diurutkan menjadi dua bagian, yang belum diurutkan (meja pertama) dan yang sudah diurutkan (meja kedua). Elemen pertama diambil dari bagian *array* yang belum diurutkan dan kemudian diletakkan sesuai posisinya pada bagian lain dari array yang telah diurutkan. Langkah ini dilakukan secara berulang hingga tidak ada lagi elemen yang tersisa pada bagian *array* yang belum diurutkan.

5.2.1 Algoritma

```
void insertionSort(Object array[], int startIdx, int endIdx) {
            for (int i = startIdx; i < endIdx; i++) {
              int k = i;
              for (int j = i + 1; j < endIdx; j++) {
                     if (((Comparable) array[k]).compareTo(array[j])>0) {
                            k = j;
                     }
              }
              swap(array[i],array[k]);
            }
}
```

6.2.2 Sebuah Contoh

Data		***1st Pass***		***2nd Pass***		***3rd Pass***		***4th Pass***
Mango		Mango		Apple		Apple		Apple
Apple		Apple		Mango		Mango		Banana
Peach		Peach		Peach		Orange		Mango
Orange		Orange		Orange		Peach		Orange
Banana		Banana		Banana		Banana		Peach

Gambar 1.1.2: Contoh insertion sort

Pada akhir modul ini, anda akan diminta untuk membuat implementasi bermacam algoritma *sorting* yang akan dibahas pada bagian ini.

5.3 *Selection Sort*

Jika anda diminta untuk membuat algoritma *sorting* tersendiri, anda mungkin akan menemukan sebuah algoritma yang mirip dengan *selection sort.* Layaknya *insertion sort*, algoritma ini sangat rapat dan mudah untuk diimplementasikan.

Mari kita kembali menelusuri bagaimana algoritma ini berfungsi terhadap satu paket kartu. Asumsikan bahwa kartu tersebut akan diurutkan secara *ascending*. Pada awalnya, kartu tersebut akan disusun secara linier pada sebuah meja dari kiri ke kanan, dan dari atas ke bawah. Pilih nilai kartu yang paling rendah, kemudian tukarkan posisi kartu ini dengan kartu yang terletak pada pojok kiri atas meja. Lalu cari kartu dengan nilai paling rendah diantara sisa kartu yang tersedia. Tukarkan kartu yang baru saja terpilih dengan kartu pada posisi kedua. Ulangi langkah – langkah tersebut hingga posisi kedua sebelum posisi terakhir dibandingkan dan dapat digeser dengan kartu yang bernilai lebih rendah.

Ide utama dari algoritma *selection sort* adalah memilih elemen dengan nilai paling rendah dan menukar elemen yang terpilih dengan elemen ke-*i*. Nilai dari *i* dimulai dari 1 ke *n*, dimana *n* adalah jumlah total elemen dikurangi 1.

5.3.1 Algoritma

```
void selectionSort(Object array[], int startIdx, int endIdx) {
            int min;
            for (int i = startIdx; i < endIdx; i++) {
                  min = i;
                  for (int j = i + 1; j < endIdx; j++) {
                        if (((Comparable)array[min]).compareTo(array[j])>0) {
                              min = j;
                        }
                  }
            swap(array[min],                                          array[i]);
            }
}
```

5.3.2 Sebuah Contoh

Data		***1st Pass***		***2nd Pass***		***3rd Pass***		***4th Pass***
Maricar		Hannah		Hannah		Hannah		Hannah
Vanessa		*Vanessa*		Margaux		Margaux		Margaux
Margaux		*Margaux*		*Vanessa*		Maricar		Maricar
Hannah		Maricar		*Maricar*		*Vanessa*		Rowena
Rowena		Rowena		Rowena		*Rowena*		Vanessa

Figure 1.2.2: Contoh selection sort

5.4 *Merge Sort*

Sebelum mendalami algoritma *merge sort*, mari kita mengetahui garis besar dari konsep *divide and conquer* karena *merge sort* mengadaptasi pola tersebut.

5.4.1 Pola Divide and Conquer

Beberapa algoritma mengimplementasikan konsep rekursi untuk menyelesaikan permasalahan. Permasalahan utama kemudian dipecah menjadi sub-masalah, kemudian solusi dari sub-masalah akan membimbing menuju solusi permasalahan utama.
Pada setiap tingkatan rekursi, pola tersebut terdiri atas 3 langkah.

1. *Divide*

Memilah masalah menjadi sub masalah

2. *Conquer*

Selesaikan sub masalah tersebut secara rekursif. Jika sub-masalah tersebut cukup

ringkas dan sederhana, pendekatan penyelesaian secara langsung akan lebih efektif

3. Kombinasi

Mengkombinasikan solusi dari sub-masalah, yang akan membimbing menuju penyelesaian atas permasalahan utama

5.4.2 Memahami Merge Sort

Seperti yang telah dijelaskan sebelumnya, *Merge sort* menggunakan pola *divide and conquer*. Dengan hal ini deskripsi dari algoritma dirumuskan dalam 3 langkah berpola *divide-and-conquer*. Berikut menjelaskan langkah kerja dari *Merge sort.*

1. *Divide*

Memilah elemen – elemen dari rangkaian data menjadi dua bagian.

2. *Conquer*

Conquer setiap bagian dengan memanggil prosedur *merge sort* secara rekursif

3. Kombinasi

Mengkombinasikan dua bagian tersebut secara rekursif untuk mendapatkan rangkaian data berurutan

Proses rekursi berhenti jika mencapai elemen dasar. Hal ini terjadi bilamana bagian yang akan diurutkan menyisakan tepat satu elemen. Sisa pengurutan satu elemen tersebut menandakan bahwa bagian tersebut telah terurut sesuai rangkaian.

5.4.3 Algoritma

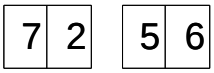
```
void mergeSort(Object array[], int startIdx, int endIdx) {
       if (array.length != 1) {
              //Membagi rangkaian data, rightArr dan leftArr
              mergeSort(leftArr, startIdx, midIdx);
              mergeSort(rightArr, midIdx+1, endIdx);
              combine(leftArr, rightArr);
       }
}
```

5.4.4 Sebuah Contoh

Rangkaian data:

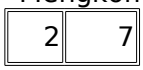
7	2	5	6

Membagi rangkaian menjadi dua bagian:
LeftArr RightArr



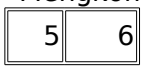
Membagi *LeftArr* menjadi dua bagian:
LeftArr RightArr

Mengkombinasikan



Membagi *RightArr* menjadi dua bagian:
LeftArr RightArr

Mengkombinasikan



Mengkombinasikan *LeftArr* dan *RightArr*.

Gambar 1.3.4: Contoh merge sort

5.5 *Quicksort*

Quicksort ditemukan oleh C.A.R Hoare. Seperti pada *merge sort*, algoritma ini juga berdasar pada pola divide-and-conquer. Berbeda dengan *merge sort*, algoritma ini hanya mengikuti langkah – langkah sebagai berikut :

1. Divide

Memilah rangkaian data menjadi dua sub-rangkaian A[p…q-1] dan A[q+1…r] dimana setiap elemen A[p…q-1] adalah kurang dari atau sama dengan A[q] dan setiap elemen pada A[q+1…r] adalah lebih besar atau sama dengan elemen pada A[q]. A[q] disebut sebagai elemen pivot. Perhitungan pada elemen q merupakan salah satu bagian dari prosedur pemisahan.

2. Conquer

Mengurutkan elemen pada sub-rangkaian secara rekursif

Pada algoritma *quicksort*, langkah ”kombinasi” tidak di lakukan karena telah terjadi pengurutan elemen – elemen pada sub-*array*

5.5.1 Algoritma

```
void quickSort(Object array[], int leftIdx, int rightIdx) {
       int pivotIdx;
       /* Kondisi Terminasi */
       if (rightIdx > leftIdx) {
              pivotIdx = partition(array, leftIdx, rightIdx);
              quickSort(array, leftIdx, pivotIdx-1);
              quickSort(array, pivotIdx+1, rightIdx);
       }
}
```

5.5.2 Sebuah Contoh

Rangkaian data:

3	1	4	1	5	9	2	6	5	3	5	8

Pilih sebuah elemen yang akan menjadi elemen pivot.

3	1	4	1	5	9	2	6	5	3	5	8

Inisialisasi elemen kiri sebagai elemen kedua dan elemen kanan sebagai elemen akhir.

	kiri										kanan
3	1	4	1	5	9	2	6	5	3	5	8

Geser elemen kiri kearah kanan sampai ditemukan nilai yang lebih besar dari elemen pivot tersebut. Geser elemen kanan ke arah kiri sampai ditemukan nilai dari elemen yang tidak lebih besar dari elemen tersebut.

		kiri							kanan		
3	1	4	1	5	9	2	6	5	3	5	8

Tukarkan antara elemen kiri dan kanan

		kiri							kanan		
3	1	3	1	5	9	2	6	5	4	5	8

Geserkan lagi elemen kiri dan kanan.

				kiri		kanan					
3	1	3	1	5	9	2	6	5	4	5	8

Tukarkan antar elemen kembali.

				kiri		kanan					
3	1	3	1	2	9	5	6	5	4	5	8

Geserkan kembali elemen kiri dan kanan.

				kanan	kiri						
3	1	3	1	2	9	5	6	5	4	5	8

Terlihat bahwa titik kanan dan kiri telah digeser sehingga mendapatkan nilai elemen kanan < elemen kiri. Dalam hal ini tukarkan elemen pivot dengan elemen kanan.

				pivot							
2	1	3	1	**3**	9	5	6	5	4	5	8

Gambar 1.4.2: Contoh quicksort

Kemudian urutkan elemen sub-rangkaian pada setiap sisi dari elemen pivot.

5.6 *Latihan*

5.6.1 Insertion Sort

Impelementasikan algoritma *insertion sort* dalam *Java* untuk mengurutkan serangkaian data *integer*. Lakukan percobaan terhadap hasil implementasi anda terhadap rangkaian data integer yang dimasukkan oleh pengguna melalui *command line*.

5.6.2 Selection Sort

Impelementasikan algoritma *selection sort* dalam Java untuk mengurutkan

serangkaian data integer. Lakukan percobaan terhadap hasil implementasi anda terhadap rangkaian data integer yang dimasukkan oleh pengguna melalui *command line*.

5.6.3 Merge Sort

Gunakan implementasi *merge sort* berikut ini terhadap serangkaian data *integer*.

```
class MergeSort {
		static void mergeSort(int array[], int startIdx,
		int endIdx) {
		if(startIdx == _____) {
			return;
		}
		int length = endIdx-startIdx+1;
		int mid = _____;
		mergeSort(array, _____, mid);
		mergeSort(array, _____, endIdx);
		int working[] = new int[length];
		for(int i = 0; i < length; i++) {
			working[i] = array[startIdx+i];
		}
		int m1 = 0;
		int m2 = mid-startIdx+1;
		for(int i = 0; i < length; i++) {
			if(m2 <= endIdx-startIdx) {
				if(m1 <= mid-startIdx) {
					if(working[m1] > working[m2]) {
						array[i+startIdx] = working[m2++];
					} else {
						array[i+startIdx] = _____;
					}
				} else {
					array[i+startIdx] = _____;
				}
			} else {
				array[_____] = working[m1++];
			}
		}
	}
		public static void main(String args[]) {
			int numArr[] = new int[args.length];
			for (int i = 0; i < args.length; i++) {
				numArr[i] = Integer.parseInt(args[i]);
			}
			mergeSort(numArr, 0, numArr.length-1);
			for (int i = 0; i < numArr.length; i++) {
				System.out.println(numArr[i]);
			}
		}
}
```

5.6.4 Quicksort

Gunakan implementasi *quicksort* berikut ini terhadap serangkaian data *integer*.

```
class QuickSort {
```

```
        static void quickSort (int[] array, int startIdx,
        int endIdx) {
               // startIdx adalah index bawah
               // endIdx is index atas
               // dari array yang akan diurutkan
               int i=startIdx, j=endIdx, h;
               //pilih elemen pertama sebagai pivot
               int pivot=array[_____];

               // memilah
                      do {
                             while (array[i]_____pivot) {
                                    i++;
                             }
                      while (array[j]>_____) {
                             j--;
                      }
                      if (i<=j) {
                             h=_____;
                             array[i]=_____;
                             array[j]=_____;
                             i++;
                             j--;
                      }
               } while (i<=j);

               // rekursi
               if (startIdx<j) {
                      quickSort(array, _____, j);
               }
               if (i<endIdx) {
                      quickSort(array, _____, endIdx);
               }
        }

        public static void main(String args[]) {
               int numArr[] = new int[args.length];
               for (int i = 0; i < args.length; i++) {
                      numArr[i] = Integer.parseInt(args[i]);
               }
               quickSort(numArr, 0, numArr.length-1);
               for (int i = 0; i < numArr.length; i++) {
                      System.out.println(numArr[i]);
               }
        }
}
```