BAB 1

Bekerja dengan Java Class Library

1.1 Tujuan

Pada sesi ini, kita akan mengantarkan beberapa konsep dasar dari *Object-Oriented objects*, 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*.

1.2 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.

1.3 *Class* dan *Object*

1.3.1 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.

1.3.2 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 1: Classs Instantiation

1.3.3 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.*

1.4 Method

1.4.1 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 membuat *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.

1.4.2 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 :

```
nameOf Object.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 );
```

1.4.3 Passing Variabel Dalam Method

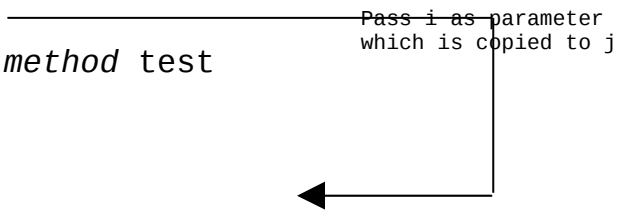
Pada contoh kita, kita telah mecoba melewati *variable* pada *method*. Bagaimanapun juga kita tidak dapat membedakan antara perbedaan tipe variabel *passing* dalam *Java*. Ada dua tipe data *passing* pada *method*, yang pertama adalah *pass-by-value* dan yang kedua adalah *pass-by-reference*.

1.4.3.1 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*

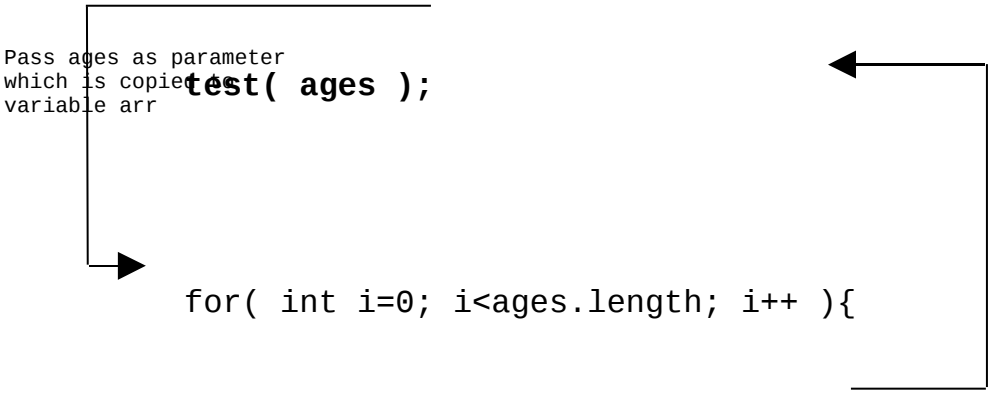
1.4.3.2 Pass-by-reference

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.

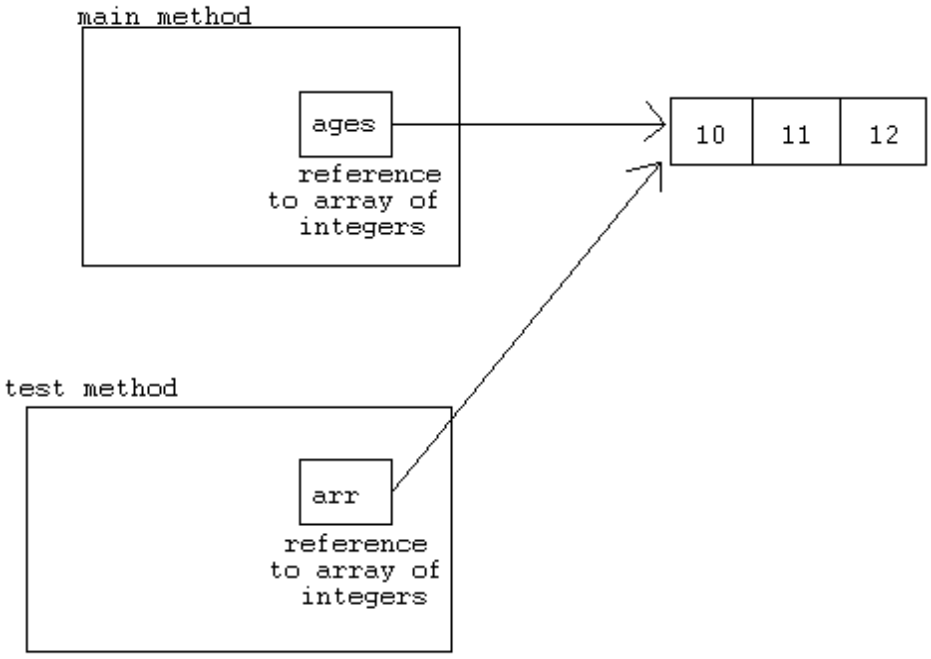
contoh :

```
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;
        }
```



```
}
}
```



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.

1.4.4 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 );
```

1.4.5 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.

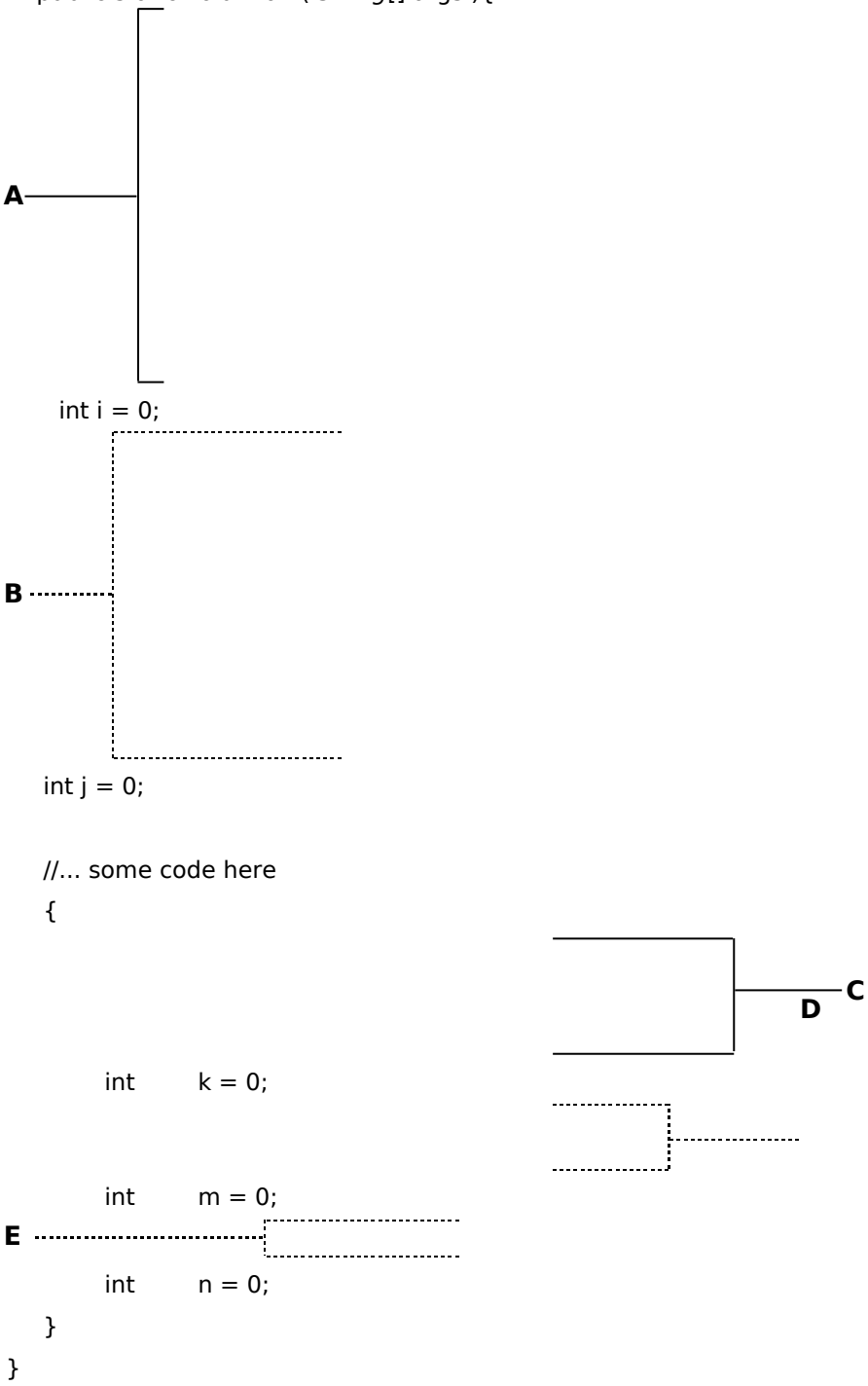
Untuk menyederhanakannya, coba berpikir tentang jangkauan apapun antara kurung kurawal {.....}, diluar kurung kurawal disebut dengan blok terluar, dan didalam kurung kurawal disebut dengan blok terdalam.

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;
      }
}
```

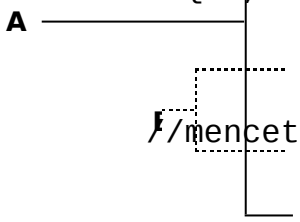


Kode yang kita miliki disini mempunyai lima jangkauan yang ditandai oleh baris dan keterangan yang mewakili jangkauan itu, dengan *variable* i,j,k,m dan n, dan 5 jangkauan A,B,C,D dan E, kita mempunyai beberapa jangkauan *variable* berikut:

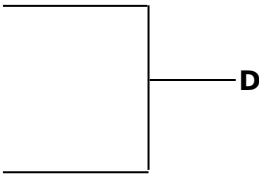
Jangkauan *variable* i adalah A.
Jangkauan *variable* j adalah B.
Jangkauan *variable* k adalah C.
Jangkauan *variable* m adalah D.
Jangkauan *variable* n adalah E.

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.

1.5 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*.

1.5.1 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
```

1.5.2 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 sub*class* 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.

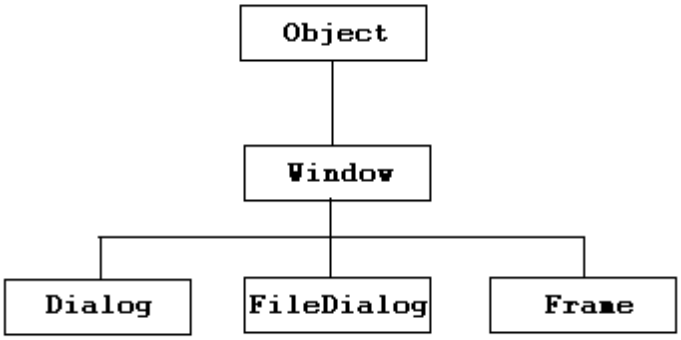


Figure 2: Contoh Hierarchy Class

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 nama*class*; *object* yang lama melanjutkan aksi seperti yang telah ada sebelumnya.

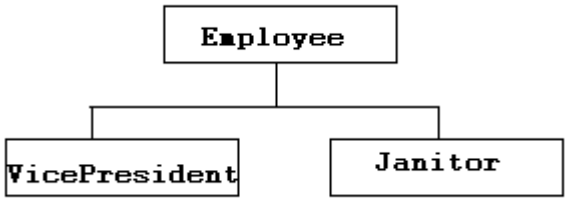


Figure 3: Class Hierarchy untuk superclass Employee

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
```

1.5.3 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.

1.5.4 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 menge*check* 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.";
```

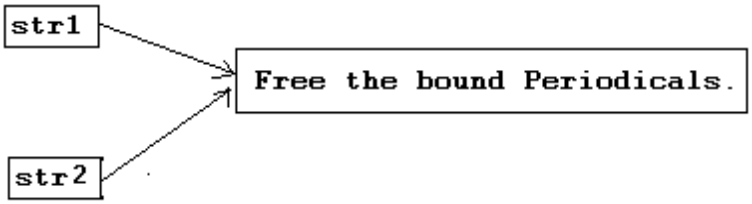


Figure 4: Keduanya mengarah ke object yang sama

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.

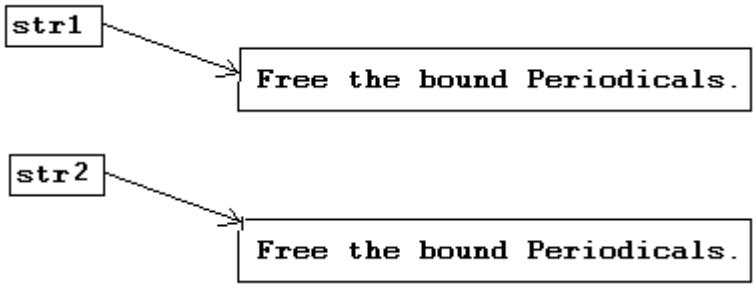


Figure 5: Sekarang mengarah pada object yang berbeda

- **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.

1.5.5 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.get*Class*().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
```

1.6 Latihan

1.6.1 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*

1.6.2 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 2

Membuat *Class* Sendiri

2.1 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*

2.2 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

Public	-	*Class* ini dapat diakses dari luar *package*
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

2.3 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.

2.3.1 ***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.

2.3.2 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
}
```

2.4. 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>[,]

2.4.1 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.*

2.4.2 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*.

2.4.3 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;
}
```

2.4.4 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.

2.4.5 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
```

2.5. 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.

2.6. *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

2.7. 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>*
}
```

2.7.1 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
}
```

2.7.2 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;
}
```

2.7.3 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
}
```

2.7.4 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

2.8. *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.

2.8.1 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;
```

2.8.2 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*.

2.8.3 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
```

2.9.*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.

2.9.1 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*.

2.9.2 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.

2.9.3 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.

2.9.4 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.

2.10. Latihan

2.10.1 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

2.10.2 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 3

Pewarisan, Polimorfisme, dan *Interface*

3.1 Tujuan

Dalam bagian ini, kita akan membicarakan bagaimana suatu *class* dapat mewariskan sifat dari *class* yang sudah ada. *Class* ini dinamakan *subclass* dan induk *class* dinamakan *superclass*. Kita juga akan membicarakan sifat khusus dari *Java* dimana kita dapat secara otomatis memakai *method* yand tepat untuk setiap *object* tanpa memperhatikan asal dari s*ubclass object*. Sifat ini dinamakan polimorfisme. Pada akhirnya, kita akan mendiskusikan tentang *interface* yang membantu mengurangi penulisan program.

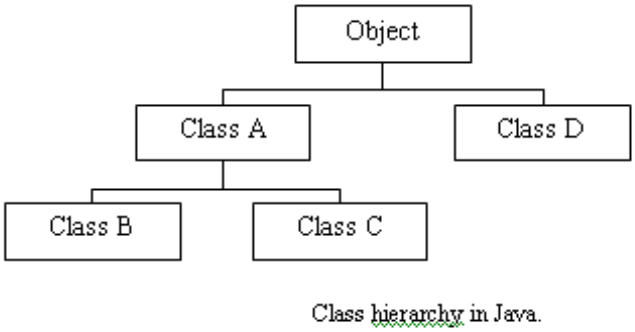
Pada akhir pembelajaran ini, siswa harus mampu untuk:

- Mendefinisikan *superclasses* dan *subclasses*
- *Override method* dari *superclasses*
- Membuat *method final* dan *class final*

3.2 Pewarisan

Dalam *Java*, semua *class*, termasuk *class* yang membangun *Java API*, adalah *subclasses* dari *superclass Object*. Contoh hirarki *class* diperlihatkan di bawah ini.

Beberapa *class* di atas *class* utama dalam hirarki *class* dikenal sebagai *superclass*. Sementara beberapa *class* di bawah *class* pokok dalam hirarki *class* dikenal sebagai *subclass* dari *class* tersebut.



Gambar 1: Hirarki *class*

Pewarisan adalah keuntungan besar dalam pemrograman berbasis *object* karena suatu sifat atau *method* didefinisikan dalam *superclass*, sifat ini secara otomatis diwariskan dari semua *subclasses*. Jadi, Anda dapat menuliskan kode *method* hanya sekali dan mereka dapat digunakan oleh semua *subclass*. *Subclass* hanya butuh mengimplementasikan perbedaannya sendiri dan induknya.

3.2.1 Mendefinisikan Superclass dan Subclass

Untuk memperoleh suatu *class*, kita menggunakan kata kunci ***extend***. Untuk mengilustrasikan ini, kita akan membuat contoh *class* induk. Dimisalkan kita mempunyai *class* induk yang dinamakan *Person*.

```
public class Person
      {

      protected String

      name;

      protected String

      address;


      /**

       * Default constructor

       */

      public Person(){


      System.out.println(“Inside Person:Constructor”);


      name = "";


      address = "";



      }


      /**

       * Constructor dengan dua parameter

       */

      public Person( String name, String address ){


      this.name = name;
```

```
this.address = address;


}

/**
 * Method accessor
 */
public String getName(){

return name;
}

public String getAddress(){

return address;
}

public void setName( String name ){

this.name = name;
}

public void setAddress( String add ){

this.address = add;
}
}
```

Perhatikan bahwa atribut *name* dan *address* dideklarasikan sebagai ***protected***. Alasannya kita melakukan ini yaitu, kita inginkan atribut-atribut ini untuk bisa diakses oleh *subclasses* dari *superclassess*. Jika kita mendeklarasikan sebagai *private*, *subclasses* tidak dapat menggunakannya. Catatan bahwa semua properti dari *superclass* yang dideklarasikan sebagai ***public, protected* dan *default*** dapat diakses oleh *subclasses*-nya.

Sekarang, kita ingin membuat *class* lain bernama *Student*. Karena *Student* juga sebagai *Person*, kita putuskan hanya meng-*extend class Person*, sehingga kita dapat mewariskan

semua properti dan *method* dari setiap *class Person* yang ada. Untuk melakukan ini, kita tulis,

```
            public class Student extends Person
            {

            public Student(){


            System.out.println("Inside Student:Constructor");


            //beberapa kode di sini

            }


            // beberapa kode di sini

      }
```

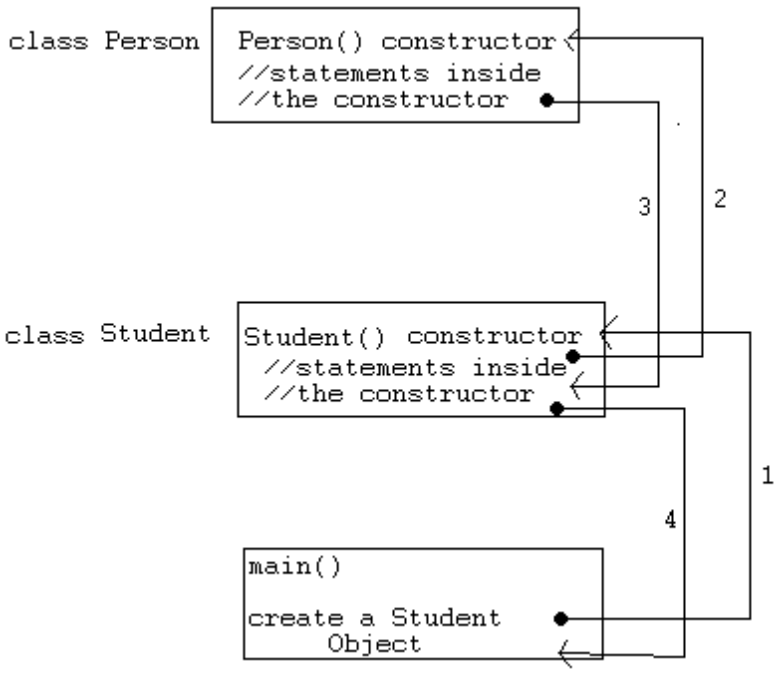
Ketika *object Student* di-*instantiate*, *default constructor* dari *superclass* secara mutlak meminta untuk melakukan inisialisasi yang seharusnya. Setelah itu, pernyataan di dalam *subclass* dieksekusi. Untuk mengilustrasikannya, perhatikan kode berikut,

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

            Student anna = new Student();

      }
```

Dalam kode ini, kita membuat sebuah *object* dari *class Student*. Keluaran dari program adalah,

```
            Inside Person:Constructor

      Inside Student:Constructor
```

Alur program ditunjukkan sebagai berikut.



Gambar 2: Alur Program

3.2.2 Kata Kunci Super

Subclass juga dapat memanggil *constructor* secara *explicit* dari *superclass* terdekat. Hal ini dilakukan dengan pemanggil construktor ***super***. Pemanggil *constructor super* dalam *constructor* dari *subclass* akan menghasilkan eksekusi dari *superclass constructor* yang bersangkutan, berdasar dari argumen sebelumnya.

Sebagai contoh, pada contoh *class* sebelumnya. *Person* dan *Student*, kita tunjukkan contoh dari pemanggil *constructor super*. Diberikan kode berikut untuk *Student*,

```
public Student(){

      super( "SomeName", "SomeAddress" );

      System.out.println("Inside Student:Constructor");

}
```

Kode ini memanggil *constructor* kedua dari *superclass* terdekat(yaitu adalah *Person*) dan mengeksekusinya. Contoh kode lain ditunjukkan sebagai berikut,

```
      public Student(){
```

```
        super();

        System.out.println("Inside Student:Constructor");

    }
```

Kode ini memanggil *default constructor* dari *superclass* terdekat(yaitu *Person*) dan mengeksekusinya.

Ada beberapa hal yang harus diingat ketika menggunakan pemanggil konstuktor *super*:

1. Pemanggil *super*() HARUS DIJADIKAN PERNYATAAN PERTAMA DALAM *constructor*.
2. Pemanggil *super*() hanya dapat digunakan dalam definisi *constructor*.
3. Termasuk *constructor this()* dan pemanggil *super()* TIDAK BOLEH TERJADI DALAM *constructor* YANG SAMA.

Pemakaian lain dari *super* adalah untuk menunjuk anggota dari *superclass*(seperti referensi ***this***). Sebagai contoh,

```
public Student()
        {

        super.name = “somename”;

        super.address = “some address”;

    }
```

3.2.3 Overriding Method

Untuk beberapa pertimbangan, kadang-kadang *class* asal perlu mempunyai implementasi berbeda dari *method* yang khusus dari *superclass* tersebut. Oleh karena itulah, *method overriding* digunakan. *Subclass* dapat mengesampingkan *method* yang didefinisikan dalam *superclass* dengan menyediakan implementasi baru dari *method* tersebut.

Misalnya kita mempunyai implementasi berikut untuk *method getName* dalam *superclass Person,*

```
        public class Person
        {

        :

        :

        public String getName(){
```

```
            System.out.println("Parent: getName");


            return name;

            }

            :
            }
```

Untuk *override, method getName* dalam *subclass Student*, kita tulis,

```
public class Student extends Person
            {

            :

            :

            public String getName(){


            System.out.println("Student: getName");


            return name;

            }

            :
            }
```

Jadi, ketika kita meminta *method getName* dari *object class Student*, *method overridde* akan dipanggil, keluarannya akan menjadi,

```
      Student: getName
```

3.2.4 Method final dan class final

Dalam *Java*, juga memungkinkan untuk mendeklarasikan *class-class* yang tidak lama menjadi *subclass*. *Class* ini dinamakan ***class final***. Untuk mendeklarasikan *class* untuk menjadi *final* kita hanya menambahkan kata kunci ***final*** dalam deklarasi *class*. Untuk contohnya, jika kita ingin *class Person* untuk dideklarasikan *final*, kita tulis,

```
      public final class Person
            {
```

```
        //area kode

    }
```

Beberapa *class* dalam *Java API* dideklarasikan secara *final* untuk memastikan sifatnya tidak dapat di-*override*. Contoh-contoh dari *class* ini adalah *Integer*, *Double*, dan *String*.

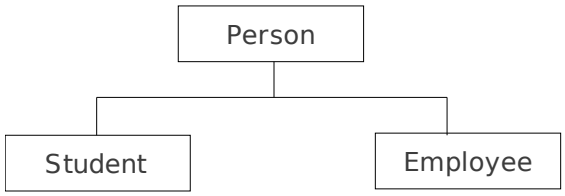
Ini memungkinkan dalam *Java* membuat *method* yang tidak dapat di-*override. Method* ini dapat kita panggil ***method final***. Untuk mendeklarasikan *method* untuk menjadi *final*, kita tambahkan kata kunci *final* ke dalam deklarasi *method*. Contohnya, jika kita ingin *method getName* dalam *class Person* untuk dideklarasikan *final*, kita tulis,

```
public final String getName(){

        return name;
        }
```

Method static juga secara otomatis *final*. Ini artinya Anda tidak dapat membuatnya *override*.

3.3 Polimorfisme

Sekarang, *class induk Person* dan *subclass Student* dari contoh sebelumnya, kita tambahkan *subclass* lain dari *Person* yaitu *Employee*. Di bawah ini adalah hierarkinya,



Gambar 3: Hirarki dari class induk Person

Dalam *Java*, kita dapat membuat referensi yang merupakan tipe dari *superclass* ke sebuah *object* dari *subclass* tersebut. Sebagai contohnya,

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

        Person

        ref;


        Student
```

```
        studentObject = new Student();

        Employee

        employeeObject = new Employee();


        ref = studentObject; //Person menunjuk kepada




            // object Student


        //beberapa kode di sini

    }
```

Sekarang dimisalkan kita punya *method getName* dalam *superclass Person* kita, dan kita *override method* ini dalam kedua *subclasses Student* dan *Employee*,

```
        public class Person
        {

        public String getName(){


        System.out.println("Person Name:" + name);


        return name;

        }
        }
        public class Student extends Person
        {

        public String getName(){


        System.out.println("Student Name:" + name);


        return name;

        }
        }
```

```
            public class Employee extends Person
            {

            public String getName(){


            System.out.println("Employee Name:" + name);


return name;


}
```

Kembali ke *method* utama kita, ketika kita mencoba memanggil *method getName* dari referensi *Person ref*, *method getName* dari *object Student* akan dipanggil. Sekarang, jika kita berikan ref ke *object Employee, method getName* dari *Employee* akan dipanggil.

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

            Person

            ref;


            Student

            studentObject = new Student();

            Employee

            employeeObject = new Employee();


            ref = studentObject; //Person menunjuk kepada


                // object Student

            String temp = ref.getName(); //getName dari Student




            //class dipanggil
```

```
            System.out.println( temp );

            ref = employeeObject; //Person menunjuk kepada




            // object Employee




            String temp = ref.getName(); //getName dari Employee




            //class dipanggil

            System.out.println( temp );

      }
```

Kemampuan dari referensi untuk mengubah sifat menurut *object* apa yang dijadikan acuan dinamakan polimorfisme. Polimorfisme menyediakan multi*object* dari *subclasses* yang berbeda untuk diperlakukan sebagai *object* dari *superclass* tunggal, secara otomatis menunjuk *method* yang tepat untuk menggunakannya ke *particular object* berdasar *subclass* yang termasuk di dalamnya.

Contoh lain yang menunjukkan properti polimorfisme adalah ketika kita mencoba melalui referensi ke *method*. Misalkan kita punya *method* statis **printInformation** yang mengakibatkan *object Person* sebagai referensi, kita dapat me-referensi dari tipe *Employee* dan tipe *Student* ke *method* ini selama itu masih *subclass* dari *class Person*.

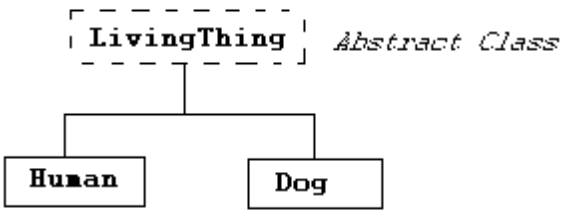
```
            public static main( String[] args )
            {

            Student

            studentObject = new Student();

            Employee

            employeeObject = new Employee();
```

```
printInformation( studentObject );

printInformation( employeeObject );
}


public static printInformation( Person p ){

. . . .
}
```

3.4 Abstract *Class*

Misalnya kita ingin membuat *superclass* yang mempunyai *method* tertentu yang berisi implementasi, dan juga beberapa *method* yang akan di-*overridden* oleh *subclasses* nya.

Sebagai contoh, kita akan membuat *superclass* bernama *LivingThing*. *class* ini mempunyai *method* tertentu seperti *breath, eat, sleep,* dan *walk*. Akan tetapi, ada beberapa *method* di dalam *superclass* yang sifatnya tidak dapat digeneralisasi. Kita ambil contoh, *method walk*. *Tidak semua kehidupan berjalan(walk) dalam cara yang sama.* Ambil manusia sebagai misal, kita manusia berjalan dengan dua kaki, dimana kehidupan lainnya seperti anjing berjalan dengan empat kaki. Akan tetapi, beberapa ciri umum dalam kehidupan sudah biasa, itulah kenapa kita inginkan membuat *superclass* umum dalam hal ini.



Gambar 4: class abstract

Kita dapat membuat *superclass* yang mempunyai beberapa *method* dengan implementasi sedangkan yang lain tidak. *Class* jenis ini yang disebut dengan *class abstract*.

Sebuah ***class abstract*** adalah *class* yang tidak dapat di-*instantiate*. Seringkali muncul di atas hirarki *class* pemrograman berbasis *object*, dan mendefinisikan keseluruhan aksi yang mungkin pada *object* dari seluruh *subclasses* dalam *class*.

Method ini dalam *class abstract* yang tidak mempunyai implementasi dinamakan *method abstract*. Untuk membuat *method abstract*, tinggal menulis deklarasi *method* tanpa tubuh *class* dan digunakan menggunakan kata kunci *abstract*. Contohnya,

```
public abstract void someMethod();
```

Sekarang mari membuat contoh *class abstract*.

```
public abstract class LivingThing
            {
            public void breath(){

            System.out.println("Living Thing breathing...");
            }

            public void eat(){

            System.out.println("Living Thing eating...");
            }

            /**
             * abstract method walk
             * Kita ingin method ini di-overridden oleh subclasses

             */
            public abstract void walk();
      }
```

Ketika *class* meng-*extend class abstract LivingThing*, dibutuhkan untuk *override method abstract walk()*, atau lainnya, juga *subclass* akan menjadi *class abstract*, oleh karena itu tidak dapat di-*instantiate*. Contohnya,

```
      public class Human extends LivingThing
```

```
        {

        public void walk(){


        System.out.println("Human walks...");

        }

    }
```

Jika *class Human* tidak dapat *overridde method walk*, kita akan menemui pesan *error* berikut ini,

```
        Human.java:1: Human is not abstract and does not override
        abstract method walk() in LivingThing
        public class Human extends LivingThing
               ^


1 error
```

Petunjuk penulisan program:
Gunakan class abstract untuk mendefinisikan secara luas sifat-sifat dari class tertinggi pada hirarki pemrograman berbasis object, dan gunakan subclassesnya untuk menyediakan rincian dari class abstract.

3.5 *Interface*

Interface adalah jenis khusus dari blok yang hanya berisi *method signature*(atau *constant*). *Interface* mendefinisikan sebuah*(signature)* dari sebuah kumpulan *method* tanpa tubuh.

Interface mendefinisikan sebuah cara standar dan umum dalam menetapkan sifat-sifat dari *class-class*. Mereka menyediakan *class-class*, tanpa memperhatikan lokasinya dalam hirarki *class*, untuk mengimplementasikan sifat-sifat yang umum. Dengan catatan bahwa *interface-interface* juga menunjukkan polimorfisme, dikarenakan program dapat memanggil *method interface* dan versi yang tepat dari *method* yang akan dieksekusi tergantung dari tipe *object* yang melewati pemanggil *method interface*.

3.5.1 Kenapa Kita Memakai Interface?

Kita akan menggunakan *interface* jika *kita ingin class yang tidak berhubungan mengimplementasikan method yang sama*. Melalui *interface-interface*, kita dapat menangkap kemiripan diantara *class* yang tidak berhubungan tanpa membuatnya seolah-olah *class* yang berhubungan.

Mari kita ambil contoh *class* ***Line*** dimana berisi *method* yang menghitung panjang dari garis

dan membandingkan *object* ***Line*** ke *object* dari *class* yang sama. Sekarang, misalkan kita punya *class* yang lain yaitu ***MyInteger*** dimana berisi *method* yang membandingkan *object* ***MyInteger*** ke *object* dari *class* yang sama. Seperti yang kita lihat disini, kedua *class-class* mempunyai *method* yang mirip dimana membandingkan mereka dari *object* lain dalam tipe yang sama, tetapi mereka tidak berhubungan sama sekali. Supaya dapat menjalankan cara untuk memastikan bahwa dua *class-class* ini mengimplementasikan beberapa *method* dengan tanda yang sama, kita dapat menggunakan sebuah *interface* untuk hal ini. Kita dapat membuat sebuah *class interface*, katakanlah *interface* ***Relation*** dimana mempunyai deklarasi *method* pembanding. Relasi *interface* dapat dideklarasikan sebagai,

```
public interface Relation
{

public boolean isGreater( Object a, Object b);

public boolean isLess( Object a, Object b);

public boolean isEqual( Object a, Object b);
}
```

Alasan lain dalam menggunakan *interface* pemrograman *object* adalah *untuk menyatakan sebuah interface* pemrograman *object tanpa menyatakan class*nya. Seperti yang dapat kita lihat nanti dalam bagian *Interface vs class*, kita dapat benar-benar menggunakan *interface* sebagai tipe data.

Pada akhirnya, kita perlu menggunakan *interface* untuk pewarisan model jamak dimana menyediakan *class* untuk mempunyai lebih dari satu *superclass*. Pewarisan jamak tidak ditunjukkan di *Java*, tetapi ditunjukkan di bahasa berorientasi *object* lain seperti C++.

3.5.2 Interface vs. Class Abstract

Berikut ini adalah perbedaan utama antara sebuah *interface* dan sebuah *class abstract*: *method interface* tidak punya tubuh, sebuah *interface* hanya dapat mendefinisikan konstanta dan *interface* tidak langsung mewariskan hubungan dengan *class* istimewa lainnya, mereka didefinisikan secara *independent*.

3.5.3 Interface vs. Class

Satu ciri umum dari sebuah *interface* dan *class* adalah pada tipe mereka berdua. Ini artinya bahwa sebuah *interface* dapat digunakan dalam tempat-tempat dimana sebuah *class* dapat digunakan. Sebagai contoh, diberikan *class Person* dan *interface PersonInterface*, berikut deklarasi yang benar:

```
PersonInterface

pi = new Person();


Person
```

```
pc = new Person();
```

Bagaimanapun, Anda tidak dapat membuat *instance* dari sebuah *interface.*
Contohnya:

```
PersonInterface

pi = new PersonInterface(); //COMPILE


//ERROR!!!
```

Ciri umum lain adalah baik *interface* maupun *class* dapat mendefinisikan *method.* Bagaimanapun, sebuah *interface* tidak punya sebuah kode implementasi sedangkan *class* memiliki salah satunya.

3.5.4 Membuat Interface

Untuk membuat *interface*, kita tulis,

```
public interface [InterfaceName]

{


//beberapa method tanpa isi


}
```

Sebagai contoh, mari kita membuat sebuah *interface* yang mendefinisikan hubungan antara dua *object* menurut urutan asli dari *object*.

```
public interface Relation
{

public boolean isGreater( Object a, Object b);

public boolean isLess( Object a, Object b);

public boolean isEqual( Object a, Object b);

}
```

Sekarang, penggunaan *interface*, kita gunakan kata kunci ***implements***. Contohnya,

```
/**
```

```
 * Class ini mendefinisikan segmen garis
 */
public class Line implements Relation
{

private double x1;

private double x2;

private double y1;

private double y2;


public Line(double x1, double x2, double y1, double y2){


this.x1 = x1;


this.x2 = x2;


this.y1 = y1;


this.y2 = y2;

}

public double getLength(){
            double length = Math.sqrt((x2-x1)*(x2-x1) +


(y2-y1)* (y2-y1));


return length;

}


public boolean isGreater( Object a, Object b){


double aLen = ((Line)a).getLength();


double bLen = ((Line)b).getLength();


return (aLen > bLen);
```

```
        }

        public boolean isLess( Object a, Object b){

        double aLen = ((Line)a).getLength();

        double bLen = ((Line)b).getLength();

        return (aLen < bLen);


        }

        public boolean isEqual( Object a, Object b){

        double aLen = ((Line)a).getLength();

        double bLen = ((Line)b).getLength();

        return (aLen == bLen);
        }


    }
```

Ketika *class* Anda mencoba mengimplementasikan sebuah *interface*, selalu pastikan bahwa Anda mengimplementasikan semua *method* dari *interface*, jika tidak, Anda akan menemukan kesalahan ini,

```
Line.java:4: Line is not abstract and does not override abstract
method

isGreater(java.lang.Object,java.lang.Object) in Relation
public class Line implements Relation
       ^

1 error
```

Petunjuk penulisan program:

Gunakan interface untuk mendefinisikan method standar yang sama dalam class-class berbeda yang memungkinkan. Sekali Anda telah membuat kumpulan definisi method standar, Anda dapat menulis method tunggal untuk memanipulasi semua class-class yang mengimplementasikan interface.

3.5.5 Hubungan dari Interface ke Class

Seperti yang telah kita lihat dalam bagian sebelumnya, *class* dapat mengimplementasikan sebuah *interface* selama kode implementasi untuk semua *method* yang didefinisikan dalam *interface* tersedia.

Hal lain yang perlu dicatat tentang hubungan antara *interface* ke *class-class* yaitu, *class* hanya dapat mengEXTEND SATU *superclass*, tetapi dapat mengIMPLEMENTASIkan BANYAK *interface*. Sebuah contoh dari sebuah *class* yang mengimplementasikan *interface* adalah,

```
public class Person implements PersonInterface,



       LivingThing,



       WhateverInterface {

       //beberapa kode di sini

}
```

Contoh lain dari *class* yang meng-*extend* satu *superclass* dan mengimplementasikan sebuah *interface* adalah,

```
public class ComputerScienceStudent extends Student



       implements PersonInterface,



            LivingThing {

       //beberapa kode di sini

}
```

Catatan bahwa sebuah *interface* bukan bagian dari hirarki pewarisan *class*. *Class* yang tidak berhubungan dapat mengimplementasikan *interface* yang sama.

3.5.6 Pewarisan Antar Interface

Interface bukan bagian dari hirarki *class*. Bagaimanapun, *interface* dapat mempunyai hubungan pewarisan antara mereka sendiri. Contohnya, misal kita punya dua *interface* **Student*Interface*** dan ***PersonInterface***. Jika *StudentInterface* meng-*extend* *PersonInterface*, maka ia akan mewariskan semua deklarasi *method* dalam *PersonInterface*.

```
            public interface PersonInterface {

            . . .
            }

            public interface StudentInterface extends PersonInterface {

            . . .

      }
```

3.6 Latihan

3.6.1 Extend StudentRecord

Dalam latihan ini, kita ingin untuk membuat catatan siswa yang lebih khusus yang berisi informasi tambahan tentang pengetahuan komputer siswa. Tugasnya adalah meng-*extend* *class StudentRecord* yang mengimplementasikan pelajaran sebelumnya. Cobalah untuk meng-*override* beberapa *method* yang ada dalam *superclass StudentRecord*, jika Anda benar-benar membutuhkannya.

3.6.2 Bentuk Abstract Class

Cobalah untuk membuat *class abstract* yang dinamai *Shape* dengan *method abstract* getArea() dan getName(). Tulis dua *subclasses*-nya yaitu *Circle* dan *Square*. Anda dapat menambahkan *method* tambahan ke dalam *subclasses* jika diinginkan.

BAB 4

Dasar Exception Handling

4.1 Tujuan

Dalam bagian ini, kita akan mempelajari teknik yang dipakai dalam *Java* dalam menangani kondisi yang tidak biasa dalam menjalankan operasi normal dalam program. Teknik ini dinamakan ***exception handling***.

Pada akhir pembelajaran, siswa mampu untuk:

- Mendefinisikan *exception*
- Menangani *exception* menggunakan blok *try-catch-finally*

4.2 Apa itu *Exception*?

Exception adalah sebuah peristiwa yang menjalankan alur proses normal pada program. Peristiwa ini biasanya berupa kesalahan*(error)* dari beberapa bentuk. Ini disebabkan program kita berakhir tidak normal.

Beberapa contoh dari *exception* yang Anda mungkin jumpai pada latihan-latihan sebelumnya adalah: *exception ArrayIndexOutOfBounds*, yang terjadi jika kita mencoba mengakses elemen *array* yang tidak ada, atau *NumberFormatException*, yang terjadi ketika kita mencoba melalui parameter bukan angka dalam *method Integer.parseInt*.

4.3 Menangani *Exception*

Untuk menangani *exception* dalam *Java*, kita gunakan blok *try-catch-finally*. Apa yang kita lakukan dalam program kita adalah kita menempatkan pernyataan yang mungkin menghasilkan *exception* dalam blok ini.

Bentuk umum dari blok *try-catch-finally* adalah,

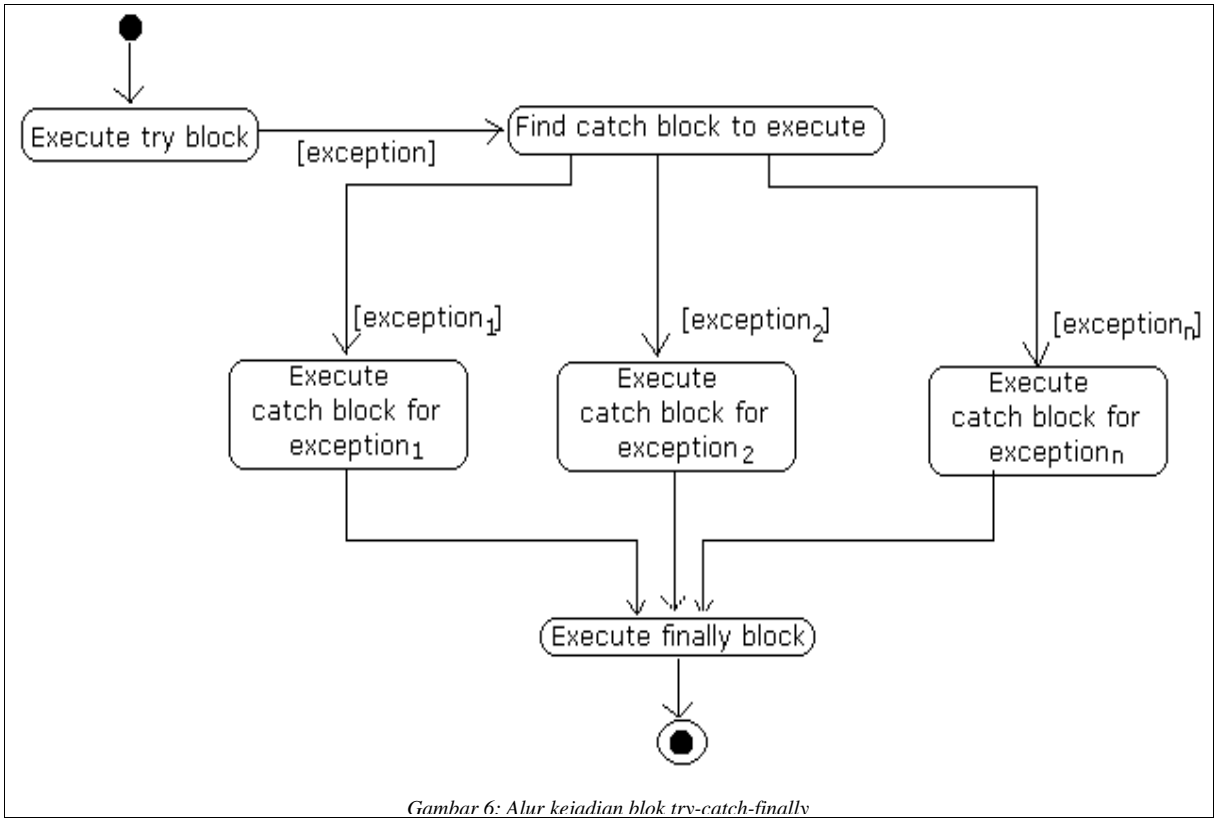
```
try{

//tulis pernyataan yang dapat mengakibatkan exception

//dalam blok ini
}
catch( <exceptionType1> <varName1> ){


//tulis aksi apa dari program Anda yang dijalankan jika ada
```

```
//exception tipe tertentu terjadi
}
. . .


catch( <exceptionType_n> <varName_n> ){

//tulis aksi apa dari program Anda yang dijalankan jika ada

//exception tipe tertentu terjadi
}
finally{

//tambahkan kode terakhir di sini
}
```

Exception dilemparkan selama eksekusi dari blok *try* dapat ditangkap dan ditangani dalam blok *catch*. Kode dalam blok *finally* selalu di-eksekusi.

Berikut ini adalah aspek kunci tentang sintak dari konstruksi *try-catch-finally*:

- Notasi blok bersifat perintah
- Setiap blok *try*, terdapat satu atau lebih blok *catch*, tetapi hanya satu blok *finally.*
- Blok *catch* dan blok *finally* harus selalu muncul dalam konjungsi dengan blok *try*, dan diatas urutan
- Blok *try* harus diikuti oleh **paling sedikit** satu blok *catch* ATAU satu blok *finally,* atau keduanya.
- Setiap blok *catch* mendefinisikan sebuah penanganan *exception*. Header dari blok *catch* harus membawa satu argumen, dimana *exception* pada blok tersebut akan ditangani. *Exception* harus menjadi *class* pelempar atau satu dari subclassesnya.



Gambar 6: Alur kejadian blok try-catch-finally

Marilah mengambil contoh kode yang mencetak argumen kedua ketika kita mencoba menjalankan kode menggunakan argumen *command-line*. Perkirakan, tidak ada pengecekan dalam kode Anda untuk angka dari argumen dan kita hanya mengakses argumen kedua args[1] segera, kita akan mendapatkan *exception* berikut.

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 1

at ExceptionExample.main(ExceptionExample.java:5)
```

Untuk mencegah kejadian ini, kita dapat menempatkan kode ke dalam blok *try-catch*. Blok *finally* hanya sebagai pilihan lain saja. Sebagai contoh, kita tidak akan menggunakan blok *finally*.

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

```
            System.out.println( args[1] );

                                                                    }catch
            ( ArrayIndexOutOfBoundsException exp ){


            System.out.println("Exception caught!");

                                                                    }
                                                                    }
        }
```

Jadi kita akan menjalankan program lagi tanpa argumen, keluarannya akan menjadi,

```
Exception caught!
```

4.4 Latihan

4.4.1 Menangkap Exception 1

Diberikan kode berikut:

```
public class TestExceptions{

        public static void main( String[] args ){

        for( int i=0; true; i++ ){

        System.out.println("args["+i+"]="+



        args[i]);

        }

        }
}
```

Compile dan jalankan program *TestExceptions*. Keluarannya akan tampak seperti ini:

```
   javac TestExceptions  one  two  three
   args[0]=one
   args[1]=two
   args[2]=three
   Exception in thread "main"

   java.lang.ArrayIndexOutOfBoundsException: 3
at TestExceptions.main(1.java:4)
```

Ubah program *TestExceptions* untuk menangani *exception*, keluaran program setelah ditangkap *exception*-nya akan seperti ini:

```
   javac TestExceptions  one  two  three
   args[0]=one
   args[1]=two
   args[2]=three
   Exception caught:

   java.lang.ArrayIndexOutOfBoundsException: 3
Quiting...
```

4.4.2 Menangkap Exception 2

Melakukan percobaan pada beberapa program yang telah Anda tulis adalah hal yang baik sebelum menghadapi *exception*. Karena pada program di atas Anda tidak menangkap *exception*, maka eksekusi dengan mudahnya berhenti mengeksekusi program Anda. Kembali kepada program diatas dan gunakan penanganan *exception*.

Bab 5
Exceptions dan Assertions

5.1 Tujuan

Dasar penanganan *exception* telah dikenalkan pada anda di kursus pemrograman pertama. Bab ini membahas secara lebih dalam mengenai *exception* dan juga sedikit menyinggung tentang *assertion*.

Setelah menyelesaikan pembahasan, anda diharapkan dapat :

1. Menangani *exception* dengan menggunakan *try, catch* dan *finally*
2. Membedakan penggunaan antara *throw* dengan *throws*
3. Menggunakan *exception class* yang berbeda – beda
4. Membedakan antara *checked exceptions* dan *unchecked exceptions*
5. Membuat *exception class* tersendiri
6. Menjelaskan keunggulan penggunaan *assertions*
7. Menggunakan *assertions*

5.2 Apa itu *Exception*?

5.2.1 Pendahuluan

Bugs dan *error* dalam sebuah program sangat sering muncul meskipun program tersebut dibuat oleh *programmer* berkemampuan tinggi. Untuk menghindari pemborosan waktu pada proses *error-checking*, *Java* menyediakan mekanisme penanganan *exception*.

Exception adalah singkatan dari *Exceptional Events*. Kesalahan *(errors)* yang terjadi saat *runtime*, menyebabkan gangguan pada alur eksekusi program. Terdapat beberapa tipe *error* yang dapat muncul. Sebagai contoh adalah *error* pembagian 0, mengakses elemen di luar jangkauan sebuah *array, input* yang tidak benar dan membuka *file* yang tidak ada.

5.2.2 Error dan Exception Classes

Seluruh *exceptions* adalah *subclasses*, baik secara langsung maupun tidak langsung, dari sebuah *root class Throwable*. Kemudian, dalam *class* ini terdapat dua kategori umum : *Error class* dan *Exception class*.

Exception class menunjukkan kondisi yang dapat diterima oleh *user* program. Umumnya hal tersebut disebabkan oleh beberapa kesalahan pada kode program. Contoh dari *exceptions* adalah pembagian oleh 0 dan *error* di luar jangkauan *array*.

Error class digunakan oleh *Java run-time* untuk menangani *error* yang muncul pada saat dijalankan. Secara umum hal ini di luar control *user* karena kemunculannya disebabkan oleh *run-time environment*. Sebagai contoh adalah *out of memory* dan *harddisk crash.*

5.2.3 Sebuah Contoh

Perhatikan contoh program berikut :

```
class DivByZero {
       public static void main(String args[]) {
              System.out.println(3/0);
              System.out.println(“Cetak.”);
       }
}
```

Jika kode tersebut dijalankan, akan didapatkan pesan kesalahan sebagai berikut :

```
Exception in thread "main" java.lang.ArithmeticException: / by
zero at DivByZero.main(DivByZero.java:3)
```

Pesan tersebut menginformasikan tipe *exception* yang terjadi pada baris dimana *exception* itu berasal. Inilah aksi *default* yang terjadi bila terjadi *exception* yang tidak tertangani. Jika tidak terdapat kode yang menangani *exception* yang terjadi, aksi *default* akan bekerja otomatis. Aksi tersebut pertama-tama akan menampilkan deskripsi *exception* yang terjadi. Kemudian akan ditampilkan *stack trace* yang mengindentifikasi *method* dimana *exception* terjadi. Pada bagian akhir, aksi *default* tersebut akan menghentikan program secara paksa.

Bagaimana jika anda ingin melakukan penanganan atas *exception* dengan cara yang berbeda? Untungnya, bahasa pemrograman *Java* memiliki 3 *keywords* penting dalam penanganan *exception*, yaitu *try, catch* dan *finally*.

5.3 Menangkap *Exception*

5.3.1 Try - Catch

Seperti yang telah dijelaskan sebelumnya, *keyword try, catch* dan *finally* digunakan dalam menangani bermacam tipe *exception*. 3 *Keyword* tersebut digunakan bersama, namun *finally* bersifat opsional. Akan lebih baik jika memfokuskan pada 2 *keyword* pertama, kemudian membahas *finally* pada bagian akhir.

Berikut ini adalah penulisan *try-catch* secara umum :

```
try {
       <code to be monitored for exceptions>
} catch (<ExceptionType1> <ObjName>) {
       <handler if ExceptionType1 occurs>
}
...
} catch (<ExceptionTypeN> <ObjName>) {
       <handler if ExceptionTypeN occurs>
}
```

Petunjuk Penulisan Program :

Blok catch dimulai setelah kurung kurawal dari kode try atau catch terkait.
Penulisan kode dalam blok mengikuti identasi

Gunakan contoh kode tersebut pada program *DivByZero* yang telah dibuat sebelumnya :

```
class DivByZero {
       public static void main(String args[]) {
              try {
                     System.out.println(3/0);
                     System.out.println("Cetak.");
              } catch (ArithmeticException exc) {
                     //Reaksi atas kejadian
                     System.out.println(exc);
              }
              System.out.println("Setelah Exception.");
       }
}
```

Kesalahan pembagian dengan bilangan 0 adalah salah satu contoh dari *ArithmethicException*. Tipe *exception* kemudian mengindikasikan klausa *catch* pada *class* ini. Program tersebut menangani kesalahan yang terjadi dengan menampilkan deskripsi dari permasalahan.

Output program saat eksekusi akan terlihat sebagai berikut :

```
java.lang.ArithmeticException: / by zero
```

```
After exception.
```

Bagian kode yang terdapat pada blok *try* dapat menyebabkan lebih dari satu tipe *exception*. Dalam hal ini, terjadinya bermacam tipe kesalahan dapat ditangani menggunakan beberapa blok *catch.* Perlu dicatat bahwa blok *try* dapat hanya menyebabkan sebuah *exception* pada satu waktu, namun dapat pula menampikan tipe *exception* yang berbeda di lain waktu.

Berikut adalah contoh kode yang menangani lebih dari satu *exception* :

```
class MultipleCatch {
       public static void main(String args[]) {
              try {
                     int den = Integer.parseInt(args[0]); //baris 4
                     System.out.println(3/den); //baris 5
              } catch (ArithmeticException exc) {
                     System.out.println("Nilai Pembagi 0.");
              } catch (ArrayIndexOutOfBoundsException exc2) {
                     System.out.println("Missing argument.");
              }
              System.out.println("After exception.");
       }
}
```

Pada contoh ini, baris ke-4 akan menghasilkan kesalahan berupa *ArrayIndexOutOfBoundsException* bilamana seorang *user* alpa dalam memasukkan *argument*, sedang baris ke-5 akan menghasilkan kesalahan *ArithmethicException* jika pengguna memasukkan nilai 0 sebagai sebuah *argument*.

Pelajari apakah yang akan terjadi terhadap program bila argumen – argumen berikut dimasukkan oleh *user* :

a) Tidak ada *argument*
b) 1
c) 0

Penggunaan *try* bersarang diperbolehkan dalam pemrograman *Java*.

```
class NestedTryDemo {
       public static void main(String args[]){
         try {
           int a = Integer.parseInt(args[0]);
           try {
             int b = Integer.parseInt(args[1]);
             System.out.println(a/b);
           } catch (ArithmeticException e) {
             System.out.println("Divide by zero error!");
                                                              }
                       } catch (ArrayIndexOutOfBoundsException) {
           System.out.println("2 parameters are required!");
                                                            }
       }
}
```

Pelajari apa yang akan terjadi pada program jika *argument* – *argument* berikut dimasukkan :

a) Tidak ada argumen
b) 15
c) 15 3

d) 15 0

Kode berikut menggunakan *try* bersarang tergabung dengan penggunaan *method*.

```
class NestedTryDemo2 {
      static void nestedTry(String args[]) {
           try {
                int a = Integer.parseInt(args[0]);
                int b = Integer.parseInt(args[1]);
                System.out.println(a/b);
           } catch (ArithmeticException e) {
                System.out.println("Divide by zero error!");
           }
      }

      public static void main(String args[]){
           try {
                 nestedTry(args);
           } catch (ArrayIndexOutOfBoundsException e) {
                 System.out.println("2 parameters are
           required!");
           }
      }
}
```

Bagaimana *output* program tersebut jika diimplementasikan terhadap *argument – argument* berikut :

a) Tidak ada argumen
b) 15
c) 15 3
d) 15 0

5.3.2 Keyword Finally

Saatnya anda mengimplementasikan *finally* dalam blok *try-cacth*. Berikut ini cara penggunaan *keyword* tersebut :

```
try {
      <kode monitor exception>
} catch (<ExceptionType1> <ObjName>) {
      <penanganan jika ExceptionType1 terjadi>
} ...
} finally {
      <kode yang akan dieksekusi saat blok try berakhir>
}
```

Petunjuk Penulisan Program :

Sekali lagi, coding convention juga mengatur penggunaan finally seperti halnya pada blok catch. Penggunaan finally dimulai setelah kurung kurawal penutup blok catch terkait. Penulisan dalam blok tersebut juga mengalami identasi.

Blok *finally* mengandung kode penanganan setelah penggunaan *try* dan *catch.* Blok kode ini selalu tereksekusi walaupun sebuah *exception* terjadi atau tidak pada blok *try.* Blok kode tersebut juga akan menghasilkan nilai *true* meskipun *return, continue* ataupun *break* tereksekusi. Terdapat 4 kemungkinan skenario yang berbeda dalam blok *try-catch-finally*. Pertama, pemaksaan keluar program terjadi bila control program dipaksa untuk melewati blok *try* menggunakan *return*, *continue* ataupun *break*. Kedua, sebuah penyelesaian normal terjadi jika *try-catch-finally* tereksekusi secara normal tanpa terjadi *error* apapun. Ketiga, kode program memiliki spesifikasi tersendiri dalam blok *catch* terhadap *exception* yang terjadi. Yang terakhir, kebalikan skenario ketiga. Dalam hal ini, *exception* yang terjadi tidak terdefinisikan pada blok *catch* manapun. Contoh dari skenario – skenario tersebut terlihat pada kode berikut ini :

```
class FinallyDemo {
     static void myMethod(int n) throws Exception{
            try {
                  switch(n) {
                        case 1: System.out.println("case pertama");

                              return;
                        case 3: System.out.println("case ketiga");
                                throw new RuntimeException("demo case
                                ketiga");
                        case 4: System.out.println("case keempat");
                                throw new Exception("demo case
                                keempat");
                        case 2: System.out.println("case Kedua");
                  }
            } catch (RuntimeException e) {
                  System.out.print("RuntimeException terjadi: ");
                  System.out.println(e.getMessage());
            } finally {
                  System.out.println("try-block entered.");
            }
     }
     public static void main(String args[]){
            for (int i=1; i<=4; i++) {
                  try {
                        FinallyDemo.myMethod(i);
                  } catch (Exception e){
                        System.out.print("Exception terjadi: ");
                        System.out.println(e.getMessage());
                  }
                  System.out.println();
            }
     }
}
```

5.4 Melempar *Exception*

5.4.1 Keyword Throw

Disamping menangkap *exception*, *Java* juga mengijinkan seorang *user* untuk melempar sebuah *exception*. Sintax pelemparan *exception* cukup sederhana.

```
throw <exception object>;
```

Perhatikan contoh berikut ini.

```
/* Melempar exception jika terjadi kesalahan input */
class ThrowDemo {
   public static void main(String args[]){
          String input = “invalid input”;
          try {
                if (input.equals(“invalid input”)) {
                      throw new RuntimeException("throw demo");
                } else {
                      System.out.println(input);
                }
                System.out.println("After throwing");
                } catch (RuntimeException e) {
                      System.out.println("Exception caught here.");
                      System.out.println(e);
          }
   }
}
```

5.4.2 Keyword Throws

Jika sebuah *method* dapat menyebabkan sebuah *exception* namun tidak menangkapnya, maka digunakan *keyword throws*. Aturan ini hanya berlaku pada *checked exception*. Anda akan mempelajari lebih lanjut tentang *checked exception* dan *unchecked exception* pada bagian selanjutnya, *“Kategori Exception”*.

Berikut ini penulisan *syntax* menggunakan *keyword throws* :

```
<type> <methodName> (<parameterList>) throws <exceptionList> {
      <methodBody>
}
```

Sebuah *method* perlu untuk menangkap ataupun mendaftar seluruh *exceptions* yang mungkin terjadi, namun hal itu dapat menghilangkan tipe *Error*, *RuntimeException*, ataupun *subclass*-nya.

Contoh berikut ini menunjukkan bahwa *method myMethod* tidak menangani *ClassNotFoundException*.

```
class ThrowingClass {
      static void myMethod() throws ClassNotFoundException {
            throw new ClassNotFoundException ("just a demo");
      }
}
```

```
class ThrowsDemo {
	public static void main(String args[]) {
		try {
			ThrowingClass.myMethod();
		} catch (ClassNotFoundException e) {
			System.out.println(e);
		}
	}
}
```

5.5 Kategori *Exception*

5.5.1 Exception Classes dan Hierarchy

Seperti yang disebutkan sebelumnya, *root class* dari seluruh *exception classes* adalah *Throwable class*. Yang disebutkan dibawah ini adalah *exception class hierarchy*. Seluruh *exceptions* ini terdefinisi pada *package java.lang.*

Exception Class Hierarchy			
Throwable	Error	LinkageError, ...	
		VirtualMachineError, ...	
	Exception	ClassNotFoundException,	
		CloneNotSupportedException,	
		IllegalAccessException,	
		InstantiationException,	
		InterruptedException,	
		IOException,	EOFException,
			FileNotFoundException,
			...
		RuntimeException,	ArithmeticException,
			ArrayStoreException,
			ClassCastException,
			IllegalArgumentException, (IllegalThreadStateException and NumberFormatException as subclasses)
			IllegalMonitorStateException,
			IndexOutOfBoundsException,
			NegativeArraySizeException,
			NullPointerException,
			SecurityException
		...	

Tabel 1.4. Exception Class Hierarchy

Sekarang anda sudah cukup familiar dengan beberapa *exception classes*, saatnya untuk mengenalkan aturan : *catch* lebih dari satu harus berurutan dari *subclass* ke *superclass.*

```
class MultipleCatchError {
	public static void main(String args[]){
		try {
			int a = Integer.parseInt(args [0]);
			int b = Integer.parseInt(args [1]);
			System.out.println(a/b);
```

```
                } catch (Exception e) {
                        System.out.println(e);
                } catch (ArrayIndexOutOfBoundsException e2) {
                        System.out.println(e2);
                }
                System.out.println("After try-catch-catch.");
        }
}
```

Setelah mengkompilasi kode tersebut akan menghasilkan pesan *error* jika *Exception class* adalah *superclass* dari *ArrayIndexOutOfBoundsException class*.

```
MultipleCatchError.java:9: exception
java.lang.ArrayIndexOutOfBoundsException has already been
caught  } catch (ArrayIndexOutOfBoundsException e2) {
```

5.5.2 Checked dan Unchecked Exceptions

Exception terdiri atas *checked* dan *unchecked exceptions*.

Checked exceptions adalah *exception* yang diperiksa oleh *Java compiler*. *Compiler* memeriksa keseluruhan program apakah menangkap atau mendaftar *exception* yang terjadi dalam *sintax throws*. Apabila *checked exception* tidak didaftar ataupun ditangkap, maka *compiler error* akan ditampilkan.

Tidak seperti *checked exceptions, unchecked exceptions* tidak berupa *compile-time checking* dalam penanganan *exceptions*. Fondasi dasar dari *unchecked exception classes* adalah *Error, RuntimeException* dan *subclass*-nya.

5.5.3 User Defined Exceptions

Meskipun beberapa *exception classes* terdapat pada *package java.lang* namun tidak mencukupi untuk menampung seluruh kemungkinan tipe *exception* yang mungkin terjadi. Sehingga sangat mungkin bahwa anda perlu untuk membuat tipe *exception* tersendiri.
Dalam pembuatan tipe *exception* anda sendiri, anda hanya perlu untuk membuat sebuah *extended class* terhadap *RuntimeException class*, maupun *Exception class* lain. Selanjutnya tergantung pada anda dalam memodifikasi *class* sesuai permasalahan yang akan diselesaikan. Members dan *constructors* dapat dimasukkan pada *exception class* milik anda.

Berikut ini contohnya :

```
class HateStringException extends RuntimeException{
        /* Tidak perlu memasukkan member ataupun construktor */
}

class TestHateString {
        public static void main(String args[]) {
                String input = "invalid input";
                try {
                        if (input.equals("invalid input")) {
                        throw new HateStringException();
```

```
                }
                        System.out.println("String accepted.");
                } catch (HateString Exception e) {
                        System.out.println("I hate this string: " + input +
                        ".");
                }
        }
}
```

5.6 *Assertions*

5.6.1 User Defined Exceptions

Assertions mengijinkan *programmer* untuk menentukan asumsi yang dihadapi. Sebagai contoh, sebuah tanggal dengan area bulan tidak berada antara 1 hingga 12 dapat diputuskan bahwa data tersebut tidak *valid*. *Programmer* dapat menentukan bulan harus berada diantara area tersebut. Meskipun hal itu dimungkinkan untuk menggunakan *contructor* lain untuk mensimulasikan fungsi dari *assertions*, namun sulit untuk dilakukan karena fitur *assertion* dapat tidak digunakan. Hal yang menarik dari *assertions* adalah seorang *user* memiliki pilihan untuk digunakan atau tidak pada saat *runtime*.

Assertion dapat diartikan sebagai extensi atas komentar yang menginformasikan pembaca kode bahwa sebagian kondisi harus terpenuhi. Dengan menggunakan *assertions*, maka tidak perlu untuk membaca keseluruhan kode melalui setiap komentar untuk mencari asumsi yang dibuat dalam kode. Namun, menjalankan program tersebut akan memberitahu anda tentang *assertion* yang dibuat benar atau salah. Jika *assertion* tersebut salah, maka *AssertionError* akan terjadi.

5.6.2 Mengaktifkan dan Menonaktifkan Exceptions

Penggunaan *assertions* tidak perlu melakukan *import package java.util.assert*. Menggunakan *assertions* lebih tepat ditujukan untuk memeriksa parameter dari *non-public methods* jika *public methods* dapat diakses oleh *class* lain. Hal itu mungkin terjadi bila penulis dari *class* lain tidak menyadari bahwa mereka dapat menonaktifkan *assertions*. Dalam hal ini program tidak dapat bekerja dengan baik. Pada *non-public methods*, hal tersebut tergunakan secara langsung oleh kode yang ditulis oleh *programmer* yang memiliki akses terhadap *methods* tersebut. Sehingga mereka menyadari bahwa saat menjalankannya, *assertion* harus dalam keadaan aktif.

Untuk mengkompilasi *file* yang menggunakan *assertions*, sebuah tambahan parameter perintah diperlukan seperti yang terlihat dibawah ini :

```
javac –source 1.4 MyProgram.java
```

Jika anda ingin untuk menjalankan program tanpa menggunakan fitur *assertions*, cukup jalankan program secara normal.

```
java MyProgram
```

Namun, jika anda ingin mengaktifkan *assertions*, anda perlu menggunakan parameter –*enableassertions* atau –*ea*.

```
java –enableassertions MyProgram
```

5.6.3 Sintax Assertions

Penulisan *assertions* memiliki dua bentuk.

Bentuk yang paling sederhana terlihat sebagai berikut :

```
assert <expression1>;
```

dimana <*expression1*> adalah kondisi dimana *assertion* bernilai *true*.

Bentuk yang lain menggunakan dua ekspresi, berikut ini cara penulisannya :

```
assert <expression1> : <expression2>;
```

dimana <*expression1*> adalah kondisi *assertion* bernilai *true* dan <expression2> adalah informasi yang membantu pemeriksaan mengapa program mengalami kesalahan.

```
class AgeAssert {
     public static void main(String args[]) {
            int age = Integer.parseInt(args[0]);
            assert(age>0);
            /* jika masukan umur benar (misal, age>0) */
            if (age >= 18) {
                  System.out.println(“Congrats! You're an adult! =)”);
            }
     }
}
```

5.7 Latihan

5.7.1 Heksadesimal ke Desimal

Tentukan sebuah angka heksadesimal sebagai *input*. Konversi angka tersebut menjadi bilangan *decimal*. Tentukan *exception class* anda sendiri dan lakukan penanganan jika *input* dari *user* bukan berupa bilangan heksadesimal.

5.7.2 Menampilkan Sebuah Berlian

Tentukan nilai *integer* positif sebagai *input*. Tampilkan sebuah berlian menggunaakan karakter asterisk (*) sesuai angka yang di*input* oleh *user*. Jika *user* memasukkan bilangan *integer* negatif, gunakan *assertions* untuk menanganinya.
Sebagai contoh, jika *user* memasukkan integer bernilai 3, program anda harus menampilkan sebuah berlian sesuai bentuk berikut :

Bab 6
Tour dari Package *java.lang*

6.1 Tujuan

Java datang dengan beberapa *class built-in* yang bermanfaat. Mari kita membahas *class-class* tersebut.

Setelah melengkapi pelajaran ini, Anda diharapkan dapat:

1. Menggunakan *class-class Java* yang telah ada
   - *Math*
   - *String*
   - *StringBuffer*
   - Wrapper
   - *Process*
   - *System*

6.2 Class Math

Java juga menyediakan konstanta dan *method* untuk menunjukkan perbedaan operasi matematika seperti fungsi trigonometri dan logaritma. Selama *method-method* ini semua *static*, Anda dapat menggunakannya tanpa memerlukan sebuah objek *Math*. Untuk melengkapi daftar konstanta dan *method-method* ini, lihatlah acuan pada dokumentasi *Java API*. Dibawah ini beberapa *method-method* umum yang sering digunakan.

***Method-Method* Math**
`public static double abs(double a)`
Menghasilkan nilai mutlak a. Sebuah *method* yang di-*overload*. Dapat juga menggunakan nilai *float* atau *integer* atau juga *long integer* sebagai parameter, dengan kondisi tipe kembalinya juga menggunakan *float* atau *integer* atau *long integer*, secara berturut-turut.
`public static double random()`
Menghasilkan nilai positif bilangan acak (random) yang lebih besar atau sama dengan 0.0 tetapi kurang dari 1.0.
`public static double max(double a, double b)`
Menghasilkan nilai maksimum, diantara dua nilai *double*, *a* and *b*. Sebuah *method* yang di-*overload*. Dapat juga menggunakan nilai *float* atau *integer* atau juga *long integer* sebagai parameter, dengan kondisi tipe kembalinya juga menggunakan *float* atau *integer* atau *long integer*, secara berturut-turut.
`public static double min(double a, double b)`
Menghasilkan nilai minimum diantara dua nilai *double*, *a* and *b*. Sebuah *method* yang di-*overload*. Dapat juga menggunakan nilai *float* atau *integer* atau juga *long integer* sebagai parameter, dengan kondisi tipe kembalinya juga menggunakan *float* atau *integer* atau *long integer*, secara berturut-turut.
`public static double ceil(double a)`
Menghasilkan bilangan bulat terkecil yang lebih besar atau sama dengan *a*.
`public static double floor(double a)`
Menghasilkan bilangan bulat terbesar yang lebih kecil atau sama dengan *a*.

`public static double exp(double a)`
Menghasilkan angka Euler, *e* pangkat a.
`public static double log(double a)`
Menghasilkan logaritma natural dari a.
`public static double pow(double a, double b)`
Menghasilkan a pangkat b.
`public static long round(double a)`
Menghasilkan pembulatan keatas ke *long* terdekat. Sebuah *method* yang di-*overload*. Dapat juga menggunakan *float* pada argument dan akan menghasilkan pembulatan ke atas ke *int* terdekat.
`public static double sqrt(double a)`
Menghasilkan akar kuadrat *a*.
`public static double sin(double a)`
Menghasilkan sinus sudut *a* dalam radian.
`public static double toDegrees(double angrad)`
Menghasilkan nilai derajat yang kira-kira setara dengan nilai radian yang diberikan.
`public static double toRadians(double angdeg)`
Menghasilkan nilai radian yang kira-kira setara dengan nilai derajat yang diberikan.

Tabel 1.1: Beberapa method dari class Math

Di bawah ini adalah program yang menunjukkan bagaimana *method-method* tersebut digunakan.

```
class MathDemo {
   public static void main(String args[]) {
      System.out.println("absolute value of -5: " +
                                                Math.abs(-5));
      System.out.println("absolute value of 5: " +
                                                Math.abs(-5));
      System.out.println("random number(max value is 10): " +
                                               Math.random()*10);
      System.out.println("max of 3.5 and 1.2: " +
                                              Math.max(3.5, 1.2));
      System.out.println("min of 3.5 and 1.2: " +
                                              Math.min(3.5, 1.2));
      System.out.println("ceiling of 3.5: " + Math.ceil(3.5));
      System.out.println("floor of 3.5: " + Math.floor(3.5));
      System.out.println("e raised to 1: " + Math.exp(1));
      System.out.println("log 10: " + Math.log(10));
      System.out.println("10 raised to 3: " + Math.pow(10,3));
      System.out.println("rounded off value of pi: " +
                                             Math.round(Math.PI));
      System.out.println("square root of 5 = " + Math.sqrt(5));
      System.out.println("10 radian = " + Math.toDegrees(10) +
                                                 " degrees");
      System.out.println("sin(90): " +
                                  Math.sin(Math.toRadians(90)));
   }
}
```

Ini adalah contoh *output* dari program yang dibuat. Coba jalankan program dan bereksperimenlah secara bebas dengan memberikan *argument*.

```
absolute value of -5: 5
absolute value of 5: 5
random number(max value is 10): 4.0855332335477605
max of 3.5 and 1.2: 3.5
min of 3.5 and 1.2: 1.2
ceiling of 3.5: 4.0
floor of 3.5: 3.0
e raised to 1: 2.7182818284590455
log 10: 2.302585092994046
10 raised to 3: 1000.0
rounded off value of pi: 3
square root of 5 = 2.23606797749979
10 radian = 572.9577951308232 degrees
sin(90): 1.0
```

6.3 Class String dan StringBuffer

Class String disediakan oleh *Java SDK* dengan menggunakan kombinasi *character literals*. Tidak seperti bahasa pemrograman lainnya,seperti C atau C++, *strings* dapat digunakan menggunakan *array* dari *character* atau disederhanakan dengan menggunakan *class String*. Sebagai catatan, bahwa sebuah objek *String* berbeda dari sebuah *array* dari *character*.

6.3.1 *Constructor String*

Class String mempunyai 11 *constructor*. Untuk melihat bagaimana *constructor- constructor* ini, perhatikan contoh berikut.

```
/* Contoh ini diambil dari catatan Dr. Encarnacion. */
class StringConstructorsDemo {
   public static void main(String args[]) {
      String s1 = new String();  // creates an empty string
      char chars[] = { 'h', 'e', 'l', 'l', 'o'};
      String s2 = new String(chars); // s2 = "hello";
      byte bytes[] = { 'w', 'o', 'r', 'l', 'd' };
      String s3 = new String(bytes);

// s3 = "world"
      String s4 = new String(chars, 1, 3);
      String s5 = new String(s2);
      String s6 = s2;
      System.out.println(s1);
      System.out.println(s2);
      System.out.println(s3);
      System.out.println(s4);
      System.out.println(s5);
      System.out.println(s6);
   }
}
```

6.3.2 *Method-method String*

Di bawah ini adalah daftar dari *method-method String*.

Method-Method String
`public char charAt(int index)`
Mengirim karakter di indeks yang dispesifikasikan oleh parameter *index*.
`public int compareTo(String anotherString)`
Membandingkan dua *String* dan mengirim bilangan int yang menspesifikasikan apakah objek *string* pemanggil kurang dari atau sama dengan *anotherString.* Bernilai negatif jika objek yang dilewatkan (*passed* string) lebih besar, 0 jika kedua string sama, dan bernilai positif jika objek string pemanggil (*calling string*) lebih besar.
`public int compareToIgnoreCase(String str)`
Serupa dengan compareTo tetapi *case insensitivity*.
`public boolean equals(Object anObject)`
Menghasilkan nilai *true* jika parameter tunggalnya tersusun dari karakter yang sama dengan objek tempat

Method-Method String
Anda memanggil *equals*. Sedangkan jika parameter yang dispesifikkan bukan sebuah objek *String* atau jika tidak cocok dengan urutan simbol pada *string*, *method* akan dikembalikan dengan nilai *false*.
`public boolean equalsIgnoreCase(String anotherString)`
Serupa dengan *equals* tetapi *case insensitivity*.
`public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`
Mendapatkan *characters* dari *string* yang dimulai pada index *srcBegin* hingga index *srcEnd* dan mengkopi character-character tersebut pada *array dst* dimulai pada index *dstBegin*.
`public int length()`
Menghasilkan panjang *String*.
`public String replace(char oldChar, char newChar)`
Mengganti karakter, semua yang kemunculan *oldChar* diganti *newChar*.
`public String substring(int beginIndex, int endIndex)`
Mengirim substring dimulai dari indeks yang dispesifikasikan *beginIndex* dan berakhir dengan indeks yang dispesifikasikan *endIndex.*
`public char[] toCharArray()`
Returns the character array equivalent of this string.
`public String trim()`
Menghilangkan *whitespace* di awal dan akhir objek *String*.
`public static String valueOf(-)`
Dapat menggunakan tipe data sederhana seperti *boolean*, *integer* atau *character*, atau juga menggunakan sebuah objek sebagai parameter. Mengirim objek *String* yang merepresentasikan tipe tertentu yang dilewatkan sebagai parameter.

Tabel 1.2.1: Beberapa method dari class String

Perhatikan bagaimana *method-method* tersebut digunakan dalam program di bawah ini.

```
class StringDemo {
   public static void main(String args[]) {
      String name = "Jonathan";
      System.out.println("name: " + name);
      System.out.println("3rd character of name: " +
                                                name.charAt(2));
      /* character yang pertama nampak secara berurutan
      mempunyai nilai unicode lebih kecil */
      System.out.println("Jonathan compared to Solomon: " +
                                      name.compareTo("Solomon"));
      System.out.println("Solomon compared to Jonathan: " +
                                  "Solomon".compareTo("Jonathan"));
      /* 'J' mempunyai nilai unicode yang lebih kecil dibanding
      'j' */
      System.out.println("Jonathan compared to jonathan: " +
                                     name.compareTo("jonathan"));
      System.out.println("Jonathan compared to jonathan (ignore
              case): " + name.compareToIgnoreCase("jonathan"));
      System.out.println("Is Jonathan equal to Jonathan? " +
```

```
                                        name.equals("Jonathan"));
        System.out.println("Is Jonathan equal to jonathan? " +
                                        name.equals("jonathan"));
        System.out.println("Is Jonathan equal to jonathan (ignore
                  case)? " + name.equalsIgnoreCase("jonathan"));
        char charArr[] = "Hi XX".toCharArray();
        /* Membutuhkan tambahan 1 untuk indeks endSrc dari
        getChars */
        "Jonathan".getChars(0, 2, charArr, 3);
        System.out.print("getChars method: ");
        System.out.println(charArr);
        System.out.println("Length of name: " + name.length());
        System.out.println("Replace a's with e's in name: " +
                                         name.replace('a', 'e'));
        /* Membutuhkan tambahan 1 untuk parameter endIndex dari
        substring*/
        System.out.println("A substring of name: " +
                                           name.substring(0, 2));
        System.out.println("Trim \"  a b c d e f   \": \"" +
                             "  a b c d e f   ".trim() + "\"");
        System.out.println("String representation of boolean
                    expression 10>10: " + String.valueOf(10>10));
        /* method toString secara implisit dipanggil method
        println */
        System.out.println("String representation of boolean
                    expression 10<10: " + (10<10));
        /* Catatan, tidak ada perubahan pada nama objek String
        meskipun setelah penggunaan semua method. */
        System.out.println("name: " + name);
    }
}
```

Ini adalah *output* dari program yang dibuat.

```
name: Jonathan
3rd character of name: n
Jonathan compared to Solomon: -9
Solomon compared to Jonathan: 9
Jonathan compared to jonathan: -32
Jonathan compared to jonathan (ignore case): 0
Is Jonathan equal to Jonathan? true
Is Jonathan equal to jonathan? false
Is Jonathan equal to jonathan (ignore case)? true
content of charArr after getChars method: Hi Jo
Length of name: 8
Replace a's with e's in name: Jonethen
A substring of name: Jo
Trim "  a b c d e f   ": "a b c d e f"
String representation of boolean expression 10>10: false
String representation of boolean expression 10<10: false
name: Jonathan
```

6.3.3 *Class StringBuffer*

Ketika objek *String* diciptakan, objek *String* tidak bisa lagi dimodifikasi. Objek *StringBuffer* serupa dengan objek *String*, kecuali kenyataan bahwa objek *StringBuffer* bersifat dapat berubah atau dapat dimodifikasi, sedangkan pada *object String* bersifat konstan. Panjang dan isi dapat diubah hingga beberapa pemanggilan *method*.

Ini adalah beberapa *method* pada *class StringBuffer*. Lihatlah acuan pada dokumentasi *Java API*.

***Method-Method* StringBuffer**
`public int capacity()`
Mengirim jumlah memori yang dialokasikan untuk *StringBuffer*.
`public StringBuffer append(-)`
Appends merepresentasikan *string* dari *argument* untuk objek *StringBuffer*. Menggunakan parameter tunggal seperti tipe-tipe data berikut: *boolean, char, char [], double, float, int, long, Object, String and StringBuffer*. Masih mempunyai versi yang di-*overload* lainnya.
`public char charAt(int index)`
Mengirim *character* di lokasi tertentu di *StringBuffer* yang dispesifikasikan parameter *index*.
`public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`
Mendapatkan *characters* dari objek yang dimulai pada indeks *srcBegin* hingga indeks *srcEnd* dan mengkopi *character- character* tersebut pada array *dst* dimulai pada indeks *dstBegin*.
`public StringBuffer delete(int start, int end)`
Menghapus *character-character* pada *range* yang ditentukan.
`public StringBuffer insert(int offset, -)`
Menyisipkan beragam tipe data di *offset* spesifik di *StringBuffer*. Sebuah *method* yang di-*overload*. Tipe data yang mungkin digunakan: *boolean, char, char [], double, float, int, long, Object and String*. Masih mempunyai versi yang di-*overload* lainnya.
`public int length()`
Memperoleh panjang atau jumlah *character* di objek *StringBuffer*.
`public StringBuffer replace(int start, int end, String str)`
Mengganti bagian dari objek, seperti yang dispesifikasikan oleh *argument* satu dua, dengan spesifikasi *string str*.
`public String substring(int start, int end)`
Substring menyaring bagian tertentu dari *string*, dimulai pada pengspesifikasian indeks *start* hingga indeks *the end*.
`public String toString()`
Mengkonversi objek ke representasi *string*.

Tabel 1.2.2: Beberapa method dari class StringBuffer

Program di bawah ini menunjukkan bagaimana menggunakan *method-method* tersebut.

```
class StringBufferDemo {
   public static void main(String args[]) {
      StringBuffer sb = new StringBuffer("Jonathan");
      System.out.println("sb = " + sb);
      /* initial capacity is 16 */
      System.out.println("capacity of sb: " + sb.capacity());
      System.out.println("append \'O\' to sb: " +
                                          sb.append("O"));
      System.out.println("sb = " + sb);
      System.out.println("3rd character of sb: " +
                                           sb.charAt(2));
      char charArr[] = "Hi XX".toCharArray();
      /* Need to add 1 to the endSrc index of getChars */
      sb.getChars(0, 2, charArr, 3);
      System.out.print("getChars method: ");
      System.out.println(charArr);
      System.out.println("Insert \'jo\' at the 3rd cell: " +
                                         sb.insert(2, "jo"));
      System.out.println("Delete \'jo\' at the 3rd cell: " +
                                            sb.delete(2,4));
      System.out.println("length of sb: " + sb.length());
      System.out.println("replace: " +
                                      sb.replace(3, 9, " Ong"));
      /* Need to add 1 to the endIndex parameter of substring*/
      System.out.println("substring (1st two characters): " +
                                         sb.substring(0, 3));
      System.out.println("implicit toString(): " + sb);
   }
}
```

Ini adalah *output* dari program yang telah dibuat di atas. Sekali lagi, bereksperimen secara bebas dengan *code-code* merupakan cara terbaik mempelajari sintaks-sintaks yang ada.

```
sb = Jonathan
capacity of sb: 24
append 'O' to sb: JonathanO
sb = JonathanO
3rd character of sb: n
getChars method: Hi Jo
Insert 'jo' at the 3rd cell: JojonathanO
Delete 'jo' at the 3rd cell: JonathanO
length of sb: 9
replace: Jon Ong
substring (1st two characters): Jon
implicit toString(): Jon Ong
```

6.4 Class-class Wrapper

Sesungguhnya, tipe data primitif seperti *int*, *char* and *long* bukanlah sebuah objek. Sehingga, variabel-variabel tipe data ini tidak dapat mengakses *method-method* dari *class Object*. Hanya objek-objek nyata, yang dideklarasikan menjadi referensi tipe data, dapat mengakses *method-method* dari *class Object*. Ada suatu keadaan, bagaimanapun, ketika Anda membutuhkan sebuah representasi objek untuk variabel-variabel tipe primitif dalam rangka menggunakan *method- method Java built*-in. Sebagai contoh, Anda boleh menambahkan variabel tipe primitif pada objek *Collection*. Disinilah *class wrapper* masuk. *Class wrapper* adalah representasi objek sederhana dari variabel-variabel non-objek yang sederhana. Demikian daftar dari *class wrapper*.

Tipe Data Primitif	***Class Wrapper yang Sesuai***
Boolean	Boolean
Char	Character
Byte	Byte
Short	Short
Int	Integer
Long	Long
Float	*Float*
Double	Double

Tabel 1.3: Tipe data primitif dan class wrappernya yang sesuai

Nama-nama *class wrapper* cukup mudah untuk diingat selama nama-nama itu sama dengan tipe data primitif. Dan juga sebagai catatan, bahwa *class-class wrapper* diawali dengan huruf besar dan versi yang ditunjukkan dari tipe data primitive.

Di bawah ini contoh penggunaan *class wrapper* untuk *boolean*.

```
class BooleanWrapper {
   public static void main(String args[]) {
      boolean booleanVar = 1>2;
      Boolean booleanObj = new Boolean("TRue");
      /* primitif ke objek; dapat juga menggunakan method
      valueOf */
      Boolean booleanObj2 = new Boolean(booleanVar);
      System.out.println("booleanVar = " + booleanVar);
      System.out.println("booleanObj = " + booleanObj);
      System.out.println("booleanObj2 = " + booleanObj2);
      System.out.println("compare 2 wrapper objects: " +
                              booleanObj.equals(booleanObj2));
      /* objek ke primitif */
      booleanVar = booleanObj.booleanValue();
      System.out.println("booleanVar = " + booleanVar);
   }
}
```

6.5 Class Process dan Runtime

6.5.1 Class Process

class Process menyediakan *method-method* untuk memanipulasi proses-proses, seperti mematikan proses, menjalankan proses dan mengecek status proses. *Class* ini merepresentasikan program- program yang berjalan. Di bawah ini beberapa *method* pada *class Process*.

***Method-Method* Process**
`public abstract void destroy()`
Mengakhiri proses.
`public abstract int waitFor() throws InterruptedException`
Tidak mengirim sampai proses yang dipanggil berakhir.

Tabel 1.4.1: Beberapa method dari class Process

6.5.2 Class Runtime

Di sisi lain, *class Runtime* merepresentasikan lingkungan *runtime*. Dua *method* penting pada *class Runtime* adalah *method getRuntime* dan *exec*.

***Method-Method* Runtime**
`public static Runtime getRuntime()`
Mengirim objek *runtime* yang merepresentasikan lingkungan *runtime* yang berhubungan dengan applikasi *Java* saat itu.
`public Process exec(String command) throws IOException`
Dikarenakan *command* yang dispesifikasikan dieksekusi. Memperbolehkan Anda mengeksekusi proses baru.

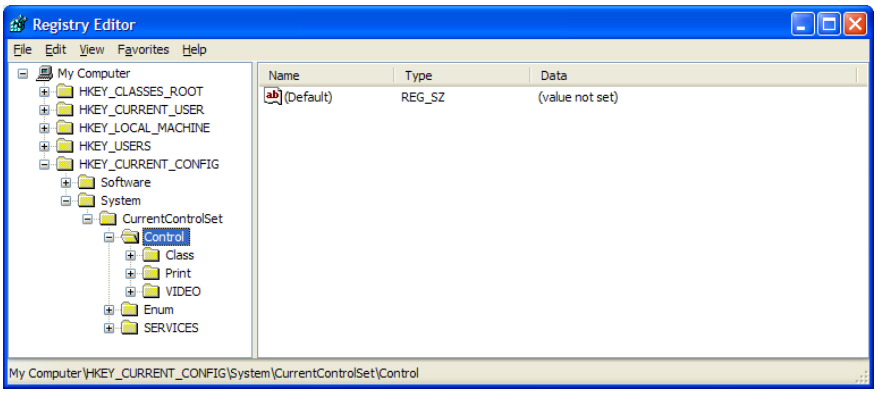
Tabel 1.4.2: Beberapa method dari class Runtime

6.5.3 Membuka Registry Editor

Berikut program untuk membuka *registry editor* tanpa harus mengetikkan perintah dari *command prompt*.

```
class RuntimeDemo {
   public static void main(String args[]) {
      Runtime rt = Runtime.getRuntime();
      Process proc;
      try {
         proc = rt.exec("regedit");
         proc.waitFor();

//try removing this line
      } catch (Exception e) {
         System.out.println("regedit is an unknown command.");
      }
   }
```

```
}
```



Gambar 1.4.3: Membuka registry editor

6.6 Class System

Class System menyediakan beberapa *field* dan *method* bermanfaat, seperti *standard input*, *standard output* dan sebuah *method* yang berguna untuk mempercepat pengkopian bagian sebuah *array*. Di bawah ini beberapa *method* menarik dari *class System*. Sebagai catatan, bahwa semua *method-method class* adalah *static*

Method-Method System
`Public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`
Mengkopi *length* elemen dari array *src* dimulai pada posisi *srcPos* ke *dest* yang dimulai pada indeks *destPos*. Lebih cepat daripada memprogram secara manual *code* untuk Anda sendiri.
`Public static long currentTimeMillis()`
Waktu dispesifikasikan dalam GMT (Greenwich Mean Time) serta merupakan jumlah milidetik yang telah dilewati sejak tengan malam 1 Januari 1970. Waktu dalam ukuran milidetik.
`Public static void exit(int status)`
Mematikan *Java Virtual Machine* (JVM) yang sedang berjalan. Nilai bukan nol untuk status konvensi yang mengindikasikan keluar yang abnormal.
`Public static void gc()`
Menjalankan *garbage collector*, yang mereklamasi space memori tak terpakai untuk digunakan kembali.
`Public static void setIn(InputStream in)`
Mengubah stream yang berhubungan dengan *System.in*, yang mana *standart* mengacu pada *keyboard*.
`Public static void setOut(PrintStream out)`
Mengubah *stream* yang berhubungan dengan *System.out*, yang mana *standart* mengacu pada *console*.

Tabel 1.5: Beberapa method dari class System

Ini adalah demo dari beberapa *method-method* tersebut.

```
import java.io.*;
```

```
class SystemDemo {
   public static void main(String args[]) throws IOException {
      int arr1[] = new int[1050000];
      int arr2[] = new int[1050000];
      long startTime, endTime;
      /* menginisialisasi arr1 */
      for (int i = 0; i < arr1.length; i++) {
         arr1[i] = i + 1;
      }
      /* mengkopi secara manual */
      startTime = System.currentTimeMillis();
      for (int i = 0; i < arr1.length; i++) {
         arr2[i] = arr1[i];
      }
      endTime = System.currentTimeMillis();
      System.out.println("Time for manual copy: " +
                                 (endTime-startTime) + " ms.");
      /* menggunakan utilitas copy yang disediakan oleh java –
         yaitu method arraycopy */
      startTime = System.currentTimeMillis();
      System.arraycopy(arr1, 0, arr2, 0, arr1.length);
      endTime = System.currentTimeMillis();
      System.out.println("Time for manual copy: " + (endTime-
                                          startTime) + " ms.");
      System.gc();

//force garbage collector to work
      System.setIn(new FileInputStream("temp.txt"));
      System.exit(0);
   }
}
```

6.7 Latihan

6.7.1 Evaluasi Ekspresi

Menggunakan *method-method class built-in Math*, buatlah sebuah program yang menggunakan nilai *double x* sebagai inputan dan evaluasilah nilai mutlak dari ekspresi yang mengikuti.
x^2 * cos(45derajat) + akar(e), e adalah angka Euler.

Input: 10
Output: 72.35939938935488
Input: 11
Output: 87.20864179427238

6.7.2 Palindrome

Palindrome adalah sebuah *string* yang membaca sama ketika mengarah ke depan atau sebaliknya. Beberapa contoh dari *palindrome* : hannah, ana, and bib. Menggunakan *String* atau *class StringBuffer*, buatlah sebuah program yang menggunakan satu *string* sebagai inputan dan tentukan jika ini sebuah *palindrome* atau bukan.

6.7.3 Notepad

Menggunakan *class Process* and *Runtime*, bukalah aplikasi *notepad* dari program *java*.

Bab 7
Teknik Pemrograman Lanjut

7.1 Tujuan

Modul ini mengenalkan suatu teknik pemrograman yang lebih tinggi. Dalam bagian ini Anda akan mempelajari rekursif dan tipe data abstrak.

Setelah menyelesaikan pelajaran ini, diharapkan Anda dapat:
1. Memahami dan menggunakan rekursif
2. Mengetahui perbedaan antara *stacks* dan *queues*
3. Mengimplementasikan suatu implementasi sequensial dari *stacks* dan *queues*
4. Mengimplementasikan suatu implementasi *linked* dari *stacks* and *queues*
5. Menggunakan *Collection classes* yang ada

7.2 Rekursif

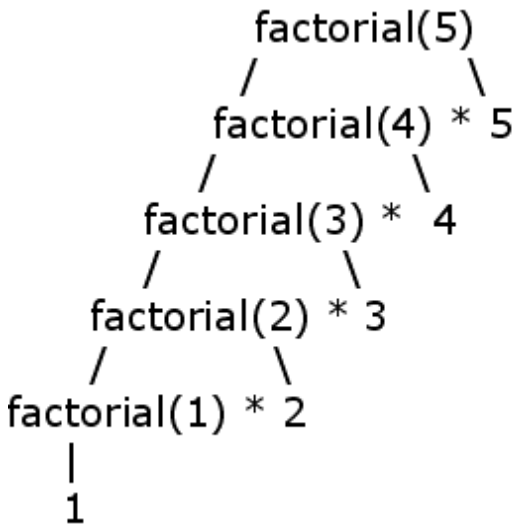
7.2.1 Apa yang dimaksud dengan Rekursif?

Rekursif adalah teknik pemecahan masalah yang *powerful* dan dapat digunakan ketika inti dari masalah terjadi berulang kali. Tentu saja, tipe dari masalah ini dapat dipecahkan mengunakan perkataan berulang-ulang (i.e., menggunakan konstruksi *looping* seperti *for, while* dan *do-while*).
Sesungguhnya, iterasi atau perkataan berulang-ulang merupakan peralatan yang lebih efisien jika dibandingkan dengan recursif tetapi *recursion* menyediakan solusi yang lebih baik untuk suatu masalah. Pada rekursif, *method* dapat memanggil dirinya sendiri. Data yang berada dalam *method* tersebut seperti *argument* disimpan sementara kedalam *stack* sampai *method* pemanggilnya diselesaikan.

7.2.2 Rekursif Vs. Iterasi

Untuk pengertian yang lebih baik dari rekursif, mari kita lihat pada bagaimana macam-macam dari teknik iterasi. Dalam teknik-teknik tersebut dapat juga kita lihat penyelesaian sebuah *loop* yang lebih baik menggunakan rekursif dari pada iterasi.

Menyelesaikan masalah dengan perulangan menggunakan iterasi secara tegas juga digunakan pada struktur kontrol pengulangan. Sementara itu, untuk rekursif, *task* diulangi dengan memanggil sebuah *method* pengulangan. Maksud dari hal tersebut adalah untuk menggambarkan sebuah masalah kedalam lingkup yang lebih kecil dari pengulangan itu sendiri. Pertimbangan suatu perhitungan yang faktorial dalam penentuan bilangan bulat. Definisi rekursif dari hal tersebut dapat diuraikan sebagai berikut: factorial(n) = factorial(n-1) * n; factorial(1) = 1. Sebagai contohnya, nilai faktorial dari 2 sama dengan fatorial (1)*2, dimana hasilnya adalah 2. Faktorial dari 3 adalah 6, dimana sama dengan faktorial dari (2)*3.



Gambar 7: Contoh Factorial

Dengan iterasi, proses diakhiri ketika kondisi *loop* gagal atau salah. Dalam kasus dari penggunaan rekursif, proses yang berakhir dengan kondisi tertentu disebut permasalahan dasar yang telah tercukupi oleh suatu pembatasan kondisi. Permasalahan yang mendasar merupakan kejadian yang paling kecil dari sebuah masalah. Sebagai contohnya, dapat dilihat pada kondisi rekursif pada faktorial, kasus yang mudah adalah ketika inputnya adalah 1. 1 dalam kasus ini merupakan inti dari masalah.

Penggunaan dari iterasi dan rekursif dapat bersama-sama memandu *loops* jika hal ini tidak digunakan dengan benar.

Keuntungan iterasi dibandingkan *recursion* adalah *performance* yang lebih baik. Hal tersebut lebih cepat untuk *recursion* sejak terbentuknya sebuah parameter pada sebuah *method* yang disebabkan oleh suatu *CPU time*. Bagaimanapun juga, rekursif mendorong *practice software engineering* yang lebih baik, sebab teknik ini biasanya dihasilkan pada kode yang singkat yang lebih mudah untuk dimengerti dan juga mempromosikan *reuseability* pada suatu solusi yang telah diterapkan.

Memilih antara iterasi dan rekursif merupakan masalah dari menjaga keseimbangan antara baiknya sebuah *performance* dan baiknya *software engineering*.

7.2.3 Factorials: Contoh

Listing program berikut ini menunjukkan bagaimana menghitung faktorial menggunakan teknik iterasi.

```
class FactorialIter {
   static int factorial(int n) {
      int result = 1;
      for (int i = n; i > 1; i--) {
         result *= i;
      }
      return result;
   }
   public static void main(String args[]) {
      int n = Integer.parseInt(args[0]);
```

```
            System.out.println(factorial(n));
         }
      }
```

Dibawah ini merupakan *listing* program yang sama tetapi menggunakan rekursif.

```
      class FactorialRecur {
         static int factorial(int n) {
            if (n == 1) {            /* The base case */
               return 1;
            }
            /* Recursive definition; Self-invocation */
            return factorial(n-1)*n;
         }
         public static void main(String args[]) {
            int n = Integer.parseInt(args[0]);
            System.out.println(factorial(n));
         }
}
```

7.2.4 Print n in any Base: Contoh yang lain

Sekarang, pertimbangan dari masalah dalam pencetakkan suatu angka desimal yang nilai *base*nya telah ditetapkan oleh pengguna. Ingat bahwa solusi dalam hal ini untuk menggunakan *repetitive division* dan untuk menulis sisa perhitungannya. Proses akan berakhir ketika sisa hasil pembagian kurang dari *base* yang ditetapkan. Dapat diasumsikan jika nilai *input* desimal adalah 10 dan kita akan mengkonversinya menjadi base 8. Inilah solusinya dengan perhitungan menggunakan pensil dan kertas.

```
8 | 10      2
  |
8 | 1       1
  |
    0
```

Dari solusi diatas, 10 adalah sama dengan 12 base 8.

Contoh berikutnya. Nilai *input* desimalnya adalah 165 dan akan dikonversi ke *base* 16.

```
16 | 165     5
   |
16 | 10      10
   |
     0
```

165 adalah sama dengan A5 base 16. Catatan: A=10.

Berikut ini merupakan solusi *iterative* untuk masalah diatas.

```
class DecToOthers {
   public static void main(String args[]) {
      int num = Integer.parseInt(args[0]);
      int base = Integer.parseInt(args[1]);
      printBase(num, base);
   }
   static void printBase(int num, int base) {
      int rem = 1;
      String digits = "0123456789abcdef";
      String result = "";
      /* the iterative step */
      while (num!=0) {
         rem = num%base;
         num = num/base;
         result = result.concat(digits.charAt(rem)+"");
      }
      /* printing the reverse of the result */
      for(int i = result.length()-1; i >= 0; i--) {
         System.out.print(result.charAt(i));
      }
   }
}
```

Berikut ini merupakan *recursion* untuk masalah yang sama dengan solusi sebelumnya.

```
class DecToOthersRecur {
   static void printBase(int num, int base) {
      String digits = "0123456789abcdef";
      /* Recursive step*/
      if (num >= base) {
         printBase(num/base, base);
      }
      /* Base case: num < base */
      System.out.print(digits.charAt(num%base));
   }
   public static void main(String args[]) {
      int num = Integer.parseInt(args[0]);
      int base = Integer.parseInt(args[1]);
      printBase(num, base);
   }


}
```

7.3 Abstract Data Type

7.3.1 Apa yang Dimaksud dengan Abstract Data Type?

Abstract Data Type (ADT) adalah kumpulan dari elemen-elemen data yang disajikan dengan satu set operasi yang digambarkan pada elemen-elemen data tersebut. *Stacks*, *queues* dan *binary trees* adalah tiga contoh dari ADT. Dalam bab ini, Anda akan mempelajari tentang *stacks* dan *queues*.

7.3.2 Stacks

Stack adalah satu *set* atau urutan elemen data dimana manipulasi data dari elemen-elemen hanya diperbolehkan pada tumpukan teratas dari *stack*. Hal ini merupakan perintah pengumpulan data secara *linier* yang disebut *"last in, first out"* (LIFO). *Stacks* berguna untuk bermacam-macam aplikasi seperti *pattern recognition* dan pengkonversian antar notasi *infix, postfix* dan *prefix* .

Dua operasi yang dihubungkan dengan *stacks* adalah operasi *push* dan *pop*. *Push* berarti memasukkan data kedalam *stacks* yang paling atas dimana pop sebagai penunjuk/*pointer* untuk memindahkan elemen ke atas *stacks*. Untuk memahami bagaimana cara kerja *stacks*, pikirkan bagaimana Anda dapat menambah atau memindakan sebuah data dari tumpukan data. Pikiran Anda akan memberitahu Anda untuk menambah atau memindahkan data hanya pada *stack* yang paling atas karena jika menggunakan cara lain, dapat menyebabkan tumpukan *stack* akan terjatuh.

Dibawah ini merupakan ilustrasi bagaimana tampilan dari *stacks*.

n-1		
...		
6		
5	Jayz	***top***
4	KC	
3	Jojo	
2	Toto	
1	Kyla	
0	DMX	***bottom***

Tabel 4.2.2: Ilustrasi Stack

Stack akan berarti penuh jika jangkauan cell teratas disimbolkan dengan n-1. Jika nilai teratas / *top* sama dengan -1, stack berarti kosong.

7.3.3 Queues

Queues adalah contoh lain dari ADT. Hal ini merupakan perintah pengumpulan data yang disebut *"first-in, first-out"*. Aplikasi ini meliputi jadwal pekerjaan dalam *operating system, topological sorting* dan *graph* traversal.

Enqueue dan *dequeue* merupakan operasi yang dihubungkan dengan *queues*. *Enqueu* menunjuk pada memasukkan data pada akhir *queue* dimana dequeue berarti memindahkan elemen dari *queue* tersebut. Untuk mengingat bagaimana *queue* bekerja, ingatlah arti khusus dari *queue* yaitu baris. Berikut ini bagaimana cara kerja *queue*. Siapa yang akan mendapatkan kesempatan pertama untuk bertemu bintang idolanya dari mereka yang sedang menunggu dalam sebuah barisan? Seharusnya orang pertama yang berada pada barisan tersebut. Orang ini mendapat kesempatan pertama untuk meninggalkan barisan. Hubungkan hal tersebut dengan bagaimana *queue* bekerja.

Berikut ini merupakan ilustrasi dari bagaimana tampilan dari *queue*.

0	1	2	3	4	5	6	7	8	9	...	n-1
		Eve	Jayz	KC	Jojo	Toto	Kyla	DMX			
		front						***end***	▪ Insert ▪ Delete		

Tabel 1.2.3: Ilustrasi Queue

Queue akan kosong jika nilai end kurang dari front. Sementara itu, akan penuh jika *end* sama dengan n-1.

7.3.4 Sequential and Linked Representation

ADTs biasanya dapat diwakilkan menggunakan *sequential* dan *linked representation*. Hal ini memudahkan untuk membuat *sequential representation* dengan menggunakan *array*. Bagaimanapun juga, masalah dengan menggunakan *array* adalah pembatasan *size*, yang membuatnya tidak fleksibel. Dengan menggunakan *array*, sering terjadi kekurangan atau kelebihan *space memory*. Mempertimbangkan hal tersebut, Anda harus membuat sebuah *array* dan mendeklarasikannya agar mampu menyimpan 50 elemen. Jika *user* hanya memasukkan 5 elemen, maka 45 *space* pada *memory* akan sia-sia. Disisi lain, jika *user* ingin memasukkan 51 elemen, *space* yang telah disediakan didalam *array* tidak akan cukup.

Dibandingkan dengan *sequential representation*, *linked representation* lebih sedikit rumit tetapi lebih fleksibel. *Linked representation* menyesuaikan *memory* yang dibutuhkan oleh *user*. Penjelasan lebih lanjut pada *linked representation* akan didiskusikan pada bab berikutnya.

7.3.5 Sequential Representation dari Integer Stack

```
class SeqStack {
   int top = -1;     /* initially, the stack is empty */
   int memSpace[];   /* storage for integers */
   int limit;        /* size of memSpace */
   SeqStack() {
      memSpace = new int[10];
      limit = 10;
   }
   SeqStack(int size) {
      memSpace = new int[size];
      limit = size;
   }
   boolean push(int value) {
      top++;
      /* check if the stack is full */
      if (top < limit) {
         memSpace[top] = value;
      } else {
         top--;
         return false;
      }
```

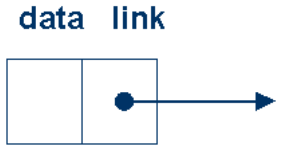
```
            return true;
        }
        int pop() {
            int temp = -1;
            /* check if the stack is empty */
            if (top >= 0) {
                temp = memSpace[top];
                top--;
            } else {
                return -1;
            }
            return temp;
        }
        public static void main(String args[]) {
            SeqStack myStack = new SeqStack(3);
            myStack.push(1);
            myStack.push(2);
            myStack.push(3);
            myStack.push(4);
            System.out.println(myStack.pop());
            System.out.println(myStack.pop());
            System.out.println(myStack.pop());
            System.out.println(myStack.pop());
        }
    }
```

7.3.6 Linked Lists

Sebelum mengimplementasikan *linked representation* dari *stacks*, pertama mari kita pelejari bagaimana membuat *linked representation*. Dalam hal ini, kita akan menggunakan *linked lists*.

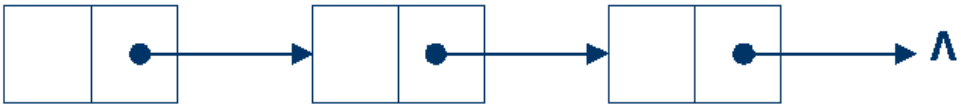
Linked list merupakan struktur dinamis yang berlawanan dengan *array*, yang merupakan struktur statis. Hal ini berarti *linked list* dapat tumbuh dan berkurang dalam *size* yang bergantung pada kebutuhan *user*. *Linked list* digambarkan sebagai kumpulan dari *nodes*, Yang masing-masing berisi data dan link atau *pointer* ke node berikutnya didalam *list*.

Gambar dibawah ini menunjukkan tampilan dari node.



Gambar 8.6a: Sebuah node

Berikut ini merupakan contoh dari *non-empty linked list* dengan 3 node.



Gambar 9.6b: Non-empty linked list dengan tiga node

Berikut ini bagaimana *class* node diimplementasikan. *Class* ini dapat digunakan untuk membuat *linked list*.

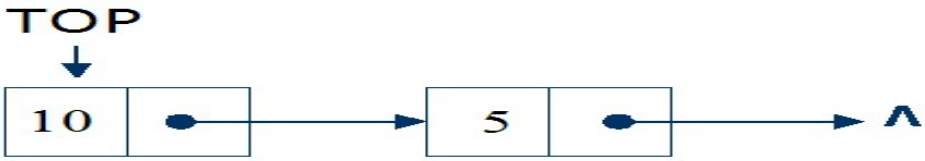
```
class Node {
   int data;        /* integer data contained in the node */
   Node nextNode;   /* the next node in the list */
}

class TestNode {
   public static void main(String args[]) {
      Node emptyList = null;    /* create an empty list */
      /* head points to 1st node in the list */
      Node head = new Node();
      /* initialize 1st node in the list */
      head.data = 5;
      head.nextNode = new Node();
      head.nextNode.data = 10;
      /* null marks the end of the list */
      head.nextNode.nextNode = null;
      /* print elements of the list */
      Node currNode = head;
      while (currNode != null) {
         System.out.println(currNode.data);
         currNode = currNode.nextNode;
      }
   }
}
```

7.3.7 Linked Representation dari Integer Stack

Sekarang Anda telah mempelajari tentang *linked list*. Maka Anda telah siap untuk menerapkan apa yang telah Anda pelajari untuk implementasi *linked representation* dari *stack*.

```
class DynamicIntStack{
   private IntStackNode top;     /* head or top of the stack */
   class IntStackNode {          /* node class */
      int data;
      IntStackNode next;
      IntStackNode(int n) {
         data = n;
         next = null;
      }
   }
```

```
    void push(int n){
       /* no need to check for overflow */
       IntStackNode node = new IntStackNode(n);
       node.next = top;
       top = node;
    }
    int pop() {
       if (isEmpty()) {
          return -1;
          /* may throw a user-defined exception */
       } else {
          int n = top.data;
          top = top.next;
          return n;
       }
    }
    boolean isEmpty(){
       return top == null;
    }
    public static void main(String args[]) {
       DynamicIntStack myStack = new DynamicIntStack();
       myStack.push(5);
       myStack.push(10);
       /* print elements of the stack */
       IntStackNode currNode = myStack.top;
       while (currNode!=null) {
          System.out.println(currNode.data);
          currNode = currNode.next;
       }
       System.out.println(myStack.pop());
       System.out.println(myStack.pop());
    }
}
```



Gambar 1.2.7: Implementasi linked dari stack

7.3.8 Java Collections

Saat ini Anda telah diperkenalkan kepada dasar *abstract data types*. Pada intinya, Anda telah mempelajari tentang dasar dari *linked lists, stacks* dan *queue*. Berita baik bahwa *abstract data types* telah siap untuk diimplementasikan dan dimasukkan dalam *Java*. Class *Stack* dan *LinkedList* diperbolehkan digunakan tanpa pengertian yang lengkap dari konsep ini. Bagaimanapun juga, sebagai ilmuwan komputer, sangat penting untuk mengerti konsep dari *abstract data types*. Oleh karena itu, penjelasan terperinci masih disampaikan dalam bagian yang terdahulu. Dengan peluncuran dari J2SE5.0, *queue*

interface telah tersedia. Untuk *detail* pada *class* dan *interface* ini, dapat dilihat pada dokumentasi Java API.

Kepada kita, Java telah menyajikan *Collection classes* dan *interfaces* yang lain, yang semuanya dapat ditemukan di *java.util* package. Contoh dari *Collection classes* termasuk *LinkedList, ArrayList, HashSet* dan *TreeSet. Class* tersebut benar-benar implementasi dari *collection interfaces* yang berbeda. Induk hirarki dari *collection interfaces* adalah *collection interfaces* itu sendiri. Sebuah *collection* hanya sebuah grup dari *object* yang diketahui sebagai elemennya sendiri. *Collection* memperbolehkan penggandaan/salinan dan tidak membutuhkan pemesanan elemen secara spesifik.

SDK tidak menyediakan implementasi *built-in* yang lain dari *interface* ini tetapi mengarahkan *subinterfaces*, *Set interfaces* dan *List interfaces* diperbolehkan. Sekarang, apa perbedaan dari kedua *interface* tersebut. *Set* merupakan *collection* yang tidak dipesan dan tidak ada penggandaan didalamnya. Sementara itu, *list* merupakan *collection* yang dipesan dari elemen-elemen dimana juga diperbolehkannya penggandaan. *HashSet, LinkedHashSet* dan *TreeSet* suatu implementasi *class* dari *Set interfaces*. *ArrayList, LinkedList* dan *Vector* suatu implementasi *class* dari *List interfaces*.

<root interface> *Collection*					
<interface> *Set*			*<interface>* *List*		
<implementing classes>			*<implementing classes>*		
HashSet	LinkedHashSet	TreeSet	ArrayList	LinkedList	Vector

Tabel 1.2.8a: Java collections

Berikut ini adalah daftar dari beberapa *Collections methods* yang disediakan dalam *Collection API* dari *Java 2 Platform* SE v1.4.1. Pada *Java 2 Platform* SE v.1.5.0, *methods* ini telah dimodifikasi untuk menampung *generic types*. Sejak *generic types* masih belum selesai dibahas, sebaiknya mempertimbangkan *method* ini terlebih dahulu. Disarankan bahwa Anda mengacu pada *Collection methods* yang terbaru dimana Anda lebih mudah mengerti *generic types*, yang akan didiskusikan pada *chapter* berikutnya.

Collection Methods
`public boolean add(Object o)`
Memasukkan *Object o* kedalam *collection* ini. Mengembalikan nilai *true* jika *o* telah sukse ditambahkan kedalam collection.
`public void clear()`
Menghapus semua element dari *collection* ini.
`public boolean remove(Object o)`
Menghapus *single instance* dari *Object o* pada *collection* ini, jika hal tersebut telah diinputkan. Mengembalikan nilai *true* jika *o* telah ditemukan dan dihapus dari *collection*.
`public boolean contains(Object o)`
Mengembalikan nilai *true* jika *collection* ini berisi *Object o*.
`public boolean isEmpty()`

Mengembalikan nilai *true* jika *collection* ini tidak berisi *object* atau element apapun.
`public int size()`
Mengembalikan nomor dari elements pada *collection* ini.
`public Iterator iterator()`
Mengembalikan sebuah *iterator* yang menunjukkan kita pada isi *collection* ini.
`public boolean equals(Object o)`
Mengembalikan nilai jika *Object o* sama dengan yang ada pada *collection*.
`public int hashCode()`
Mengembalikan nilai *hash code* (i.e., the ID) untuk *collection* ini. *Objects* atau *collections* yang sama memiliki nilai *hash code* atau ID yang sama.

Tabel 1.2.8b: Methods dari class Collection

Anda diharapkan mengacu pada dokumentasi API untuk mengetahui daftar lengkap dari *methods* dalam *Collection, List* dan *Set interface*.
Saat ini kita akan melihat beberapa *collection classes*. Harap mengacu pada API untuk daftar dari *methods* yang dimasukkan kedalam *class* ini.

Pada bagian sebelumnya, Anda telah melihat bagaimana mengimplementasikan *linked list* dengan cara Anda sendiri. *Java SDK* juga telah menyediakan *built-implementation* dari *linked list* untuk kita. *LinkedList class* berisi *methods* yang memperbolehkan *linked list* digunakan seperti *stacks, queue* atau ADT yang lain. *Listing* program berikut ini menunjukan bagaimana menggunakan *class LinkedList*.

```
import java.util.*;

class LinkedListDemo {
   public static void main(String args[]) {
      LinkedList list = new LinkedList();
      list.add(new Integer(1));
      list.add(new Integer(2));
      list.add(new Integer(3));
      list.add(new Integer(1));
      System.out.println(list + ", size = " + list.size());
      list.addFirst(new Integer(0));
      list.addLast(new Integer(4));
      System.out.println(list);
      System.out.println(list.getFirst() + ", " +
                                          list.getLast());
      System.out.println(list.get(2) + ", " + list.get(3));
      list.removeFirst();
      list.removeLast();
      System.out.println(list);
      list.remove(new Integer(1));
      System.out.println(list);
      list.remove(3);
      System.out.println(list);
```

```
      list.set(2, "one");
      System.out.println(list);
   }
}
```

ArrayList merupakan versi fleksibel dari *array* biasa. Yang mengimplementasikan *List interface*. Telitilah kode berikut ini.

```
import java.util.*;

class ArrayListDemo {
   public static void main(String args[]) {
      ArrayList al = new ArrayList(2);
      System.out.println(al + ", size = " + al.size());
      al.add("R");
      al.add("U");
      al.add("O");
      System.out.println(al + ", size = " + al.size());
      al.remove("U");
      System.out.println(al + ", size = " + al.size());
      ListIterator li = al.listIterator();
      while (li.hasNext())
         System.out.println(li.next());
      Object a[] = al.toArray();
      for (int i=0; i<a.length; i++)
         System.out.println(a[i]);
   }
}
```

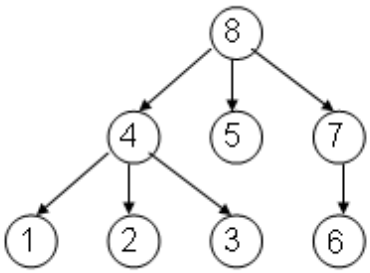
HashSet merupakan sebuah implementasi dari *Set interface* yang berguna pada hash *table*. Penggunaan suatu *hash table* lebih mudah dan cepat untuk melihat lebih *detail* elemen-elemen yang ada. *Table* menggunakan suatu rumusan untuk menentukan dimana suatu objek disimpan. Teliti program ini, yang menggunakan *class HashSet*.

```
import java.util.*;

class HashSetDemo {
   public static void main(String args[]) {
      HashSet hs = new HashSet(5, 0.5f);
      System.out.println(hs.add("one"));
      System.out.println(hs.add("two"));
      System.out.println(hs.add("one"));
      System.out.println(hs.add("three"));
      System.out.println(hs.add("four"));
      System.out.println(hs.add("five"));
      System.out.println(hs);
   }
}
```

TreeSet merupakan sebuah implementasi dari *Set interface* yang menggunakan *tree*. *Class* ini memastikan bahwa yang disortir akan diurutkan secara *ascending*.

Pertimbangkan, bagaimana *class TreeSet* telah digunakan dalam *listing* program berikut ini.

```
import java.util.*;

class TreeSetDemo {
   public static void main(String args[]) {
      TreeSet ts = new TreeSet();
      ts.add("one");
      ts.add("two");
      ts.add("three");
      ts.add("four");
      System.out.println(ts);
   }
}
```



Gambar 1.2.8: Contoh TreeSet

7.4 Latihan

7.4.1 Faktor Persekutuan Terbesar

Faktor persekutuan terbesar (FPB) dari dua angka adalah angka yang terbesar selalu dibagi oleh angka yang satunya, kemudian modulus atau sisa pembagian membagi angka kedua dan seterusnya hingga sisa pembagian dari kedua angka tersebut sama dengan nol. Menggunakan *Euclid's method,* buatlah dua kode untuk penghitungan dua angka. Gunakan iterasi untuk kode program yang pertama dan rekursif untuk kode program berikutnya.
Catatan pada algoritma Euclid :

1. Sebagai masukkan integers x dan y.
2. Ulangi step dibawah ini while y != 0
    a. y = x % y;
    b. x = Nilai lama y;
3. Return x.

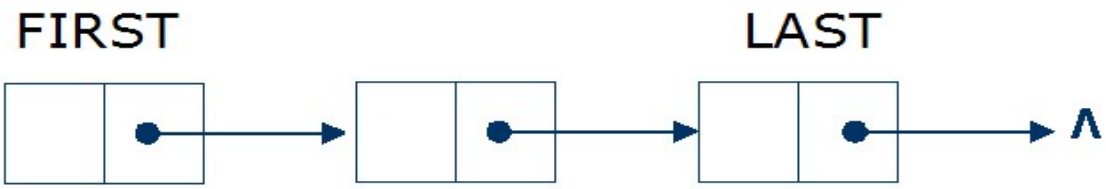
Contoh, x = 14 dan y = 6.
y = x % y = 14 % 6 = 2
x = 6
y = x % y = 6 % 2 = 0
x = 2 (FPB)

7.4.2 Sequential Representation dari Integer Queue

Dengan menggunakan *array*, implementasikan sebuah *integer queue* seperti contoh pada *sequential stack*.

7.4.3 Linked Representation dari Integer Queue

Dengan menggunakan ide dari *linked list*, implementasikan sebuah *integer queue* dinamis seperti *integer stack* dinamis yang diperkenalkan seperti contoh berikut.



7.4.4 Address Book

Dengan menggunakan *Java collection*, buatlah sebuah program yang memperbolehkan *user* untuk *insert, delete* dan *view address*. Setiap *address* berisi nama , alamat dan nomor telepon dari orang yang mengisinya. Pengisian data dimasukkan dengan cara *queue* tetapi penghapusan dilakukan dengan cara *stack*.
Dalam contoh ini, kita akan menggunakan *text* editor untuk mengedit program *Java*. Juga membuka terminal *window* untuk meng-*compile* dan meng-*execute* program *Java* anda.