1 PERANCANGAN PERANGKAT LUNAK BERORIENTASI OBYEK

1.1 Pendahuluan

Software engineering merupakan sebuah disiplin ilmu yang cukup pelik. Selama 50 tahun terakhir, para ilmuwan komputer berusaha menciptakan sistem perangkat lunak yang mudah digunakan dengan cara menggunakan kode program yang dapat dipakai kembali (*reusable code*). Pada mulanya, mereka menggunakan bahasa komputer untuk menyembunyikan kompleksitas yang ada pada bahasa mesin dan menambahkan sistem operasi yang dapat dipanggil untuk menangani operasi-operasi standar seperti membuka, membaca, dan menulis file.

Kelompok pengembang perangkat lunak lainnya mengelompokkan fungsi dan prosedur sejenis ke dalam pustaka (*library*) yang jenisnya bervariasi, mulai dari pustaka untuk menghitung beban struktur bangunan (NASTRAN), menulis karakter dan byte ke dalam komputer lain melalui jaringan (TCP/IP), membuat tampilan jendela, dan lainnya. Sebagian besar dari pustaka-pustaka tersebut memanipulasi data dalam bentuk struktur data record yang terbuka, seperti menggunakan struktur dalam bahasa C. Masalah yang timbul dari struktur record yang demikian adalah perancang pustaka tidak dapat menyembunyikan data yang digunakan dalam sebuah prosedur. Hal ini menyebabkan sulitnya memodifikasi implementasi pustaka dengan tidak mempengaruhi kode program *client* yang menggunakannya.

Pada akhir tahun 80-an, bahasa pemrograman C++ mulai memperkenalkan pemrograman berorientasi obyek (*Object-Oriented Programming*/OOP). Keunggulan dari OOP adalah aspek-aspek tertentu dari implementasi pustaka dapat disembunyikan sehingga proses update tidak mempengaruhi kode program klien (dengan mengasumsikan bahwa *interface* yang digunakan pustaka tidak berubah). Keunggulan lainnya adalah prosedur-prosedur yang ada diasosiasikan dengan struktur data. Kombinasi dari atribut data dan prosedur atau *method* ini dikenal dengan nama kelas.

Hal yang setara dengan pustaka fungsi (*function libraries*) sekarang ini adalah pustaka kelas (*class libraries*) atau *toolkits*. Pustaka-pustaka ini memungkinkan kelas untuk melakukan banyak fungsi seperti halnya pustaka fungsi. Tidak hanya itu saja, dengan menggunakan *subclassing*, *programmer* dapat dengan mudah memperluas kemampuan *tools* ini sesuai dengan keinginan mereka.

Framework menyediakan pustaka yang dapat diimplementasikan oleh berbagai *vendor*. Dengan adanya *framework* sangat dimungkinkan untuk memilih tingkat fleksibilitas dan kinerja yang sesuai dengan aplikasi yang dikembangkan.

1.1.1 Fase analisis dan disain

Ada lima tahapan di dalam proses pengembangan perangkat lunak, yaitu pendefinisian masalah, analisis, disain, implementasi dan pengujian. Kelima hal tersebut memiliki peranan penting, namun harus dipastikan bahwa ada cukup waktu untuk proses analisis dan disain.

Dalam tahap analisis, didefinisikan apa yang ingin dicapai oleh sistem. Hal ini dilakukan dengan mendefinisikan aktor dan aktivitas. Tahapan ini juga harus dapat mengidentifikasi obyek domain (*domain object*) baik yang bersifat fisik maupun konseptual yang akan dimanipulasi oleh sistem selain juga perilaku dan interaksi antar obyek. Perilaku dalam obyek-obyek ini mengimplementasi aktivitas-aktivitas yang ada. Hal yang patut diingat dalam pembuatan deskripsi aktivitas adalah ia harus dirancang dengan cukup lengkap untuk membuat kriteria awal dalam tahap pengujian.

Dalam tahap disain, didefinisikan bagaimana sistem akan mencapai tujuannya. Tahapan ini meliputi pemodelan aktor, aktivitas, obyek, dan perilaku sistem. *Unified Modeling Language* (UML) biasanya digunakan sebagai *tools* pemodelannya.

1.1.2 Abstraksi

Proses disain perangkat lunak telah beralih dari pengembangan di tingkat *low-level*, seperti menulis kode dalam bahasa mesin menuju ke level yang lebih tinggi. Ada dua hal yang mengiringi proses tersebut, yaitu penyederhanaan dan abstraksi. Penyederhanaan terjadi ketika pendisain bahasa pemrograman di tahap awal membangun konstruksi bahasa tingkat tinggi, seperti *statement* IF dan *loop* FOR dari bahasa mesin. Sedangkan abstraksi adalah penyembunyian detil implementasi yang sifatnya *private* ke dalam *interface* publik.

Konsep abstraksi ini mengarah kepada penggunaan sub rutin atau fungsi dalam bahasa pemrograman tingkat tinggi dan pengelompokan fungsi dan data ke dalam obyek. Di tingkat yang lebih tinggi, ia mengarah kepada pengembangan *framework* dan pustaka (API),

1.1.3 Kelas sebagai cetak biru dari obyek

Seorang ahli gambar membuat cetak biru (*blueprint*) untuk sebuah alat yang dapat digunakan berulang kali. Hal serupa berlaku dalam kelas. Kelas merupakan sebuah *blueprint* dari perangkat lunak yang dapat digunakan untuk menginstantiasi (menciptakan) banyak obyek. Sebuah kelas mendefinisikan satu set elemen data (yang biasa disebut dengan atribut) dan perilaku atau fungsi (yang biasa disebut dengan *method*) yang digunakan untuk memanipulasi obyek atau melakukan interaksi antar obyek. Atribut dan *method* ini secara bersama-sama dikenal sebagai anggota obyek.

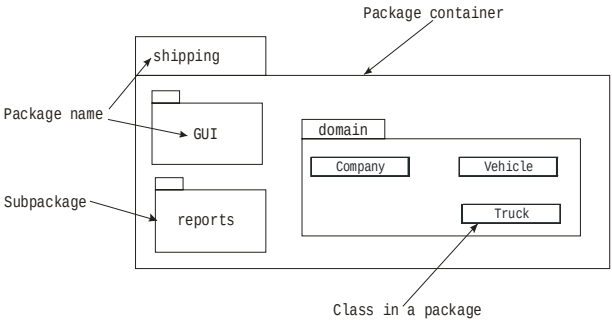
1.2 Pengenalan UML

Bahasa pemodelan yang terpadu (*Unified Modeling Language*/UML) merupakan bahasa berbasis grafis yang digunakan untuk memodelkan sistem perangkat lunak. Bahasa ini pertama-tama dikembangkan oleh tiga orang ahli pemodelan obyek, yaitu Grady Booch, James Rumbaugh, dan Ivars Jacobson pada awal tahun 90-an. Sekarang, bahasa UML telah menjadi standar untuk pemodelan software berbasis obyek.

Dalam buku ini, hanya akan dibahas secara singkat pemakaian bahasa UML yang memiliki kaitan dengan materi di dalam buku ini.

1.2.1 *Package*

Di dalam UML, *package* mengelompokkan elemen-elemen pemodelan di dalam grup. *Package* dalam UML ini dapat digunakan untuk memodelkan *package* di dalam bahasa Java. Berikut ini merupakan contoh dari *package* :



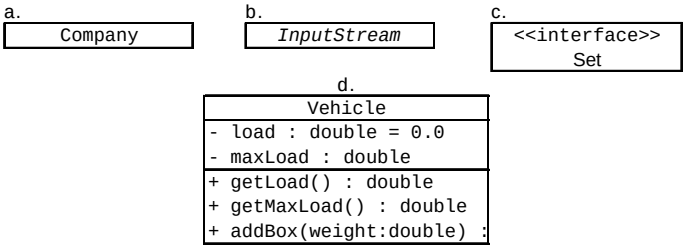
Gambar 1.1 Contoh *package*.

1.2.2 *Class diagram*

Class diagram merepresentasikan struktur statik dari sebuah sistem, di mana diagram ini terdiri dari kelas-kelas dan hubungan yang ada di antaranya.

1.2.2.1 *Class node*

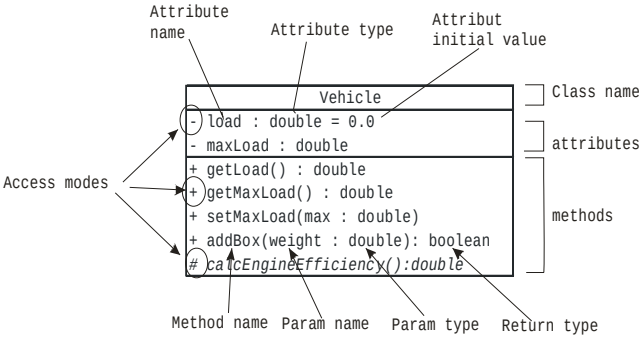
Class node merupakan representasi grafis dari sebuah kelas. Gambar di bawah ini adalah contoh dari *class node*, di mana ia berupa nama kelas yang diletakkan di dalam kotak. Contoh (a), (b), dan (c) merupakan contoh *class node* yang hanya berupa nama kelasnya saja. (Contoh (a) merupakan bentuk dari kelas konkrit yang anggotanya tidak turut dimodelkan. Contoh (b) merupakan kelas abstrak dengan nama kelas yang dicetak miring dan contoh (c) merupakan sebuah *interface*) (konsep akan kelas abstrak dan *interface* akan dibahas kemudian). Sementara contoh (d) menggambarkan sebuah kelas lengkap dengan atribut dan methodnya.



Gambar 1.2 Contoh *class node.*

Sebuah bentuk *class node* yang lengkap diperlihatkan dalam Gambar 1.3. *Class node* berbentuk sebuah kotak yang dibagi menjadi tiga bagian. Bagian pertama berisikan nama kelas, bagian kedua berisi atribut, dan bagian ketiga berisi *method-method* dalam kelas.

Pada bagian deklarasi atribut (bagian kedua), sebuah atribut dicirikan dengan elemen mode akses, nama atribut, tipe data, dan nilai inisialnya. Pemberian nilai inisial ini bersifat opsional. Sedangkan, pada bagian deklarasi *method*, sebuah *method* dicirikan dengan elemen mode akses, nama *method*, nama parameter, tipe parameter, dan tipe kembalian dari *method*.



Gambar 1.3 Contoh *class node* yang lengkap.

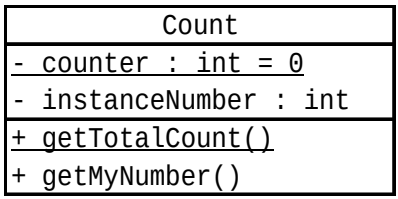
Simbol-simbol yang digunakan sebagai mode akses dalam UML adalah sebagai berikut :

Tabel 1.1
Simbol untuk Mode Akses

Mode akses	Simbol
private	-
protected	#
public	+

Konsep tentang mode akses sendiri akan dibahas kemudian. Hal yang perlu diperhatikan adalah untuk mode akses *default*, tidak digunakan simbol apapun (*blank*).

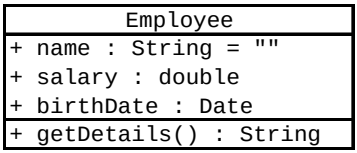
Gambar berikut ini merupakan sebuah contoh dari *class node* yang memiliki elemen statik. Hal ini ditunjukkan dengan garis bawah pada elemen bersangkutan. Sebagai contoh, *counter* merupakan data atribut `static` dan `getTotalCount` merupakan *method* `static`.



Gambar 1.4 Contoh *member* statik dalam *class node*.

1.3 *Subclassing*

Dalam pemrograman, seringkali dibuatkan suatu model untuk suatu obyek (model untuk seorang karyawan misalnya), dan membutuhkan tipe yang lebih spesifik dari model awal tersebut. Sebagai contoh, model seorang *manager*. Seorang manajer adalah karyawan plus beberapa atribut tambahan. Misalnya, dibuatkan sebuah model UML seorang karyawan (*Employee*) dalam bentuk kelas sebagai berikut :

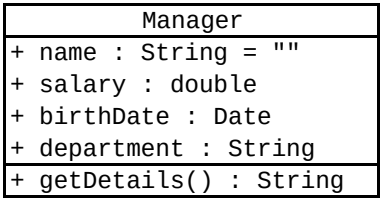


Gambar 1.5 Disain kelas Employee.

Dalam bentuk kode bahasa Java, bagan di atas dapat diterjemahkan sebagai berikut:

```
public class Employee {
         public String name = “”;
         public double salary;
         public Date birthDate;

         public String getDetails() {…}
}
```

Selain itu, dapat juga dibuatkan sebuah model UML untuk seorang *Manager* dengan bentuk sebagai berikut:



Gambar 1.6 Disain kelas Manager.

Model Manager di atas dalam kode bahasa Java diterjemahkan sebagai berikut :

```
public class Manager {
        public String name = “”;
        public double salary;
        public Date birthDate;
        public String department;

        public String getDetails() {…}
}
```

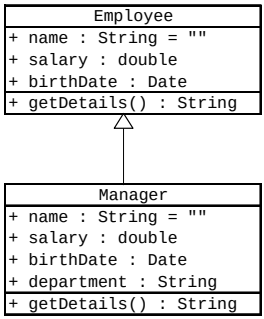
Contoh di atas menggambarkan penduplikasian data antara kelas Manager dan kelas Employee. Di dalam kelas Employee terdapat atribut name, salary, dan birthDate. Begitu juga di dalam kelas Manager. Selain itu, method getDetails() juga terdapat baik dalam kelas Employee maupun Manager.

Hal di atas adalah contoh disain yang tidak baik. Untuk membuat disain kelas yang lebih baik, perlu dibuatkan sebuah kelas baru dari kelas yang sudah ada, dalam kasus ini membuat kelas Manager dari kelas Employee. Proses ini dinamakan *subclassing* atau *inheritance* (pewarisan).

Dalam bahasa pemrograman Java, proses *subclassing* dicapai dengan menggunakan kata kunci extends, seperti terlihat dalam contoh penggalan program berikut :

```
public class Manager extends Employee {
        public String department;
}
```

Dalam bentuk diagram UML, hal tersebut akan tergambar sebagai berikut :



Gambar 1.7 Hubungan antara kelas Employee dan Manager.

Dalam contoh ini, kelas `Manager` disebut sebagai *subclass* dan `Employee` sebagai *superclass*. Panah yang menunjuk ke kelas `Employee` menggambarkan bahwa kelas `Manager` adalah turunan atau *subclass* dari kelas `Employee`.

1.4 *Access Control*

Access control berlaku pada kelas dan *class member* (meliputi atribut dan *method*).

1.4.1 *Access control* pada kelas

Ketika kode dalam satu kelas (kelas A) memiliki akses ke kelas yang lain (kelas B), hal tersebut berarti kelas A dapat melakukan satu dari tiga hal berikut:

- Membuat *instance* kelas B.
- Meng-*extends* kelas B (dengan kata lain, menjadi *subclass* dari kelas A.
- Mengakses atribut dan method tertentu dalam kelas B tergantung dari *access control* atribut dan methodnya

Access control yang dapat diterapkan pada kelas adalah akses `default` dan `public`.

1.4.1.1 *Access `default`* pada kelas

Dalam deklarasinya, sebuah kelas dengan *access control default* tidak memiliki *modifier* di depan nama kelasnya. Kelas yang memiliki akses *default* berarti kelas ini hanya dapat diakses oleh kelas yang berada di dalam *package* yang sama. Sebagai contoh, jika kelas A dan B terletak di dalam *package* yang berbeda, dan kelas A memiliki akses *default*, maka kelas B tidak dapat membuat instance dari kelas A, atau bahkan mendeklarasikan variabel yang bertipe kelas A. Perhatikan contoh berikut ini :

```
package packageOne;
 class Animal {
 }
```

Selanjutnya didefinisikan lain sebagai berikut :

```
package packageTwo;
import packageOne.Animal;
class Dog extends Animal {
}
```

Superclass Animal berada di dalam *package* yang berbeda dengan *subclass*nya, yaitu Dog. Pernyataan import di dalam kelas Dog berusaha mengimpor kelas Animal. Ketika proses kompilasi dijalankan, kelas Animal dapat dikompilasi, tetapi tidak dengan kelas Dog. Hal ini disebabkan karena *superclass* Animal mempunyai akses *default* dan terletak di dalam *package* yang berbeda.

1.4.1.2 *Access public* pada kelas

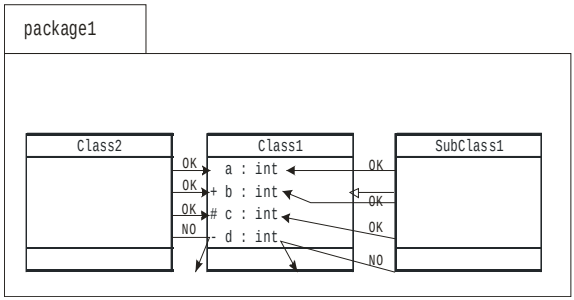
Kelas yang dideklarasikan public dapat diakses oleh kelas yang berada di manapun, termasuk oleh kelas yang berada di *package* yang berbeda. Jika contoh kode di atas dimodifikasi dengan menjadikan kelas Animal memiliki *access control* public, maka kedua kelas tersebut dapat dikompilasi.

```
package packageOne;
public class Animal {//public modifier
}
```

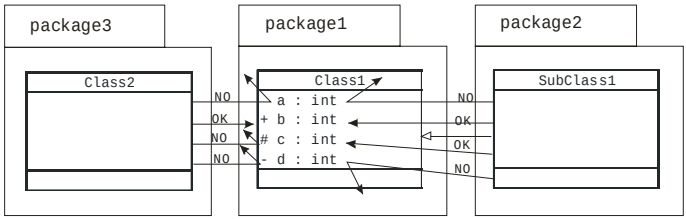
Dengan modifikasi tersebut, semua kelas dapat menginstantiasi kelas Animal dan meng-*extends* kelas ini – kecuali kelas Animal diberi modifier final (pembahasan tentang modifier final ada pada bagian selenjutnya).

1.4.2 *Access control* pada *class member*

Class member dapat memiliki satu dari empat *access control*, yaitu public, protected, default, dan private. Perhatikan contoh di bawah ini :



Gambar 1.8 Akses *class member* dari *package* yang sama.



Gambar 1.9 Akses *class member* dari package yang berbeda.

1.4.2.1 *Access control* `private` pada *class member*

Sebuah variabel atau *method* yang dideklarasikan `private` hanya dapat diakses oleh *method* yang merupakan *member* dari kelas tersebut. Ia tidak dapat diakses oleh kelas lain yang berada di dalam *package* yang sama ataupun di lain *package*.

Sebagai contoh dalam gambar 1.8, atribut `d` dalam kelas `Class1` dideklarasikan `private`. Akibatnya, kelas `Class2` yang berada dalam *package* yang sama (package1) tidak dapat mengakses atribut tersebut. Begitu juga dengan kelas `SubClass1` yang merupakan *subclass* dari kelas `Class1` tidak dapat mengakses variabel d tersebut.

Dalam gambar 1.9, variabel `d` dalam kelas `Class1` juga dideklarasikan `private`. Kelas `Class2` ataupun kelas `SubClass1` yang berada di dalam *package* yang berbeda tidak dapat mengakses atribut ini.

1.4.2.2 *Access control* `default` pada *class member*

Sama dengan *access control default* dalam kelas, sebuah variabel atau *method* mempunyai hak akses `default` jika ia tidak dituliskan secara eksplisit dalam deklarasi. Hak akses yang demikian berarti akses hanya diizinkan bagi kelas yang berada di dalam *package* yang sama.

Dalam gambar 1.8, atribut a dalam kelas `Class1` dideklarasikan *default*. Akibatnya, kelas `Class2` yang berada dalam *package* yang sama (package1) dapat mengakses atribut tersebut. Begitu juga dengan kelas `SubClass1` yang merupakan *subclass* dari kelas `Class1` dapat mengakses variabel d tersebut.

Dalam gambar 1.9, variabel a dalam kelas `Class1` juga dideklarasikan *default*. Kelas `Class2` ataupun kelas `SubClass1` tidak dapat mengakses atribut ini karena mereka berada di dalam *package* yang berbeda.

1.4.2.3 *Access control* `protected` pada *class member*

Access control `protected` berarti *member* dapat diakses oleh kelas yang berada dalam *package* yang sama dan *subclass* yang berada di dalam *package* yang berbeda.

Dalam gambar 1.8, atribut `c` dalam kelas `Class1` dideklarasikan `protected`. Akibatnya, kelas `Class2` yang berada dalam *package* yang sama (package1) dapat mengakses atribut tersebut. Begitu juga dengan kelas `SubClass1` yang merupakan *subclass* dari kelas `Class1` dapat mengakses variabel d tersebut.

Dalam gambar 1.9, variabel `c` dalam kelas `Class1` juga di-*mark* `protected`. Kelas `Class2` tidak dapat mengakses atribut ini karena ia berada di *package* yang berbeda. Tetapi `SubClass1` dapat mengakses atribut ini karena meskipun ia berada di dalam *package* yang berbeda, ia merupakan *subclass* dari kelas `Class1`.

1.4.2.4 *Access control* `public` pada *class member*

Sebuah variabel atau method yang dideklarasikan `public` dapat diakses oleh kelas manapun, apakah kelas tersebut berada di dalam *package* yang sama atau berbeda.

Dalam gambar 1.8, atribut b dalam kelas `Class1` dideklarasikan `public`. Kelas yang berada dalam *package* yang sama (`Class2` dan `SubClass1`) dapat mengakses atribut tersebut.

Dalam gambar 1.9, variabel b dalam kelas `Class1` juga dideklarasikan `public`. Kelas `Class2` dan `SubClass1` yang berada di dalam *package* yang berbeda dapat mengakses variabel b.

Tabel di bawah ini merupakan rangkuman dari *access control* pada *class member*.

Tabel 1.2
Aksesibilitas untuk *Class Member*

Modifier	Kelas yang sama	*Package* yang Sama	Subclass	Semua Kelas
`private`	Ya			
`default`	Ya	Ya		
`protected`	Ya	Ya	Ya	
`public`	Ya	Ya	Ya	Ya

1.5 *Overriding Method*

Selain dapat membuat kelas dari kelas yang sudah ada dengan menambahkan fitur tambahan, dapat juga dimodifikasi perilaku kelas induk atau *superclass*.

Sebuah method dikatakan meng-*override* method di kelas induknya jika di dalam *subclass* didefinisikan *method* yang memiliki nama, tipe kembalian dan daftar argumen yang persis sama. Contoh berikut menggambarkan konsep *overriding* :

```
public class Employee {
      protected String name = “”;
      protected double salary;
      protected Date birthDate;

      public String getDetails() {
         return “Name: ” + name + “\n” +
                “Salary: ” + salary;
      }
}

public class Manager extends Employee {
      protected String department;

      public String getDetails() {
         return “Name: ” + name + “\n” +
                “Salary: ” + salary + “\n” +
                “Manager of: ” + department;
      }
}
```

Secara definitif, kelas `Manager` mempunya method yang bernama `getDetails()` karena kelas `Manager` mewarisinya dari kelas `Employee`. Tetapi, kelas `Manager` mengubah (meng-*override*) isi dari method tersebut.

1.5.1 Aturan tentang *method* yang meng-*override*

Aturan dalam meng-*override* method dari kelas induk adalah sebagai berikut :

1. Daftar argumen pada *method* harus sama (jumlah argumen dan tipenya) dengan method yang di-*override*.
2. Tipe kembalian *method* harus sama dengan *method* yang di-*override*.
3. *access control method* tidak boleh lebih ketat (*restrictive*) daripada method yang di-*override*.
4. method yang meng-*override* tidak dapat melempar *exception* di luar *exception* yang dideklarasikan pada method yang di-*override*. Materi tentang *exception* dibahas pada bab selanjutnya.

Untuk lebih jelasnya, perhatikan contoh berikut:

```
class Parent {
  public void doSomething() {}
}

class Child extends Parent {
  private void doSomething() {}
}

public class UseBoth {
  public void doOtherThing() {
    Parent p1 = new Parent();
    Parent p2 = new Child();

    p1.doSomething();
    p2.doSomething();
  }
}
```

Di dalam kelas Parent, dideklarasikan *method* doSomething() dengan *access control* public. Lalu di kelas Child yang merupakan *subclass* dari kelas Parent, didekalarasikan *method* dengan nama, daftar argumen, dan tipe kembalian yang sama, tetapi dengan *access control* yang lebih ketat (private). Selanjutnya, dibuatkan obyek Child yang dimasukkan ke dalam variabel bertipe Parent pada baris ke-12. Di baris ke-15 dipanggil method doSomething() (p2.doSomething()) . Menurut aturan dalam program Java, method doSomething() dari kelas Child lah yang akan dieksekusi, tetapi method tersebut dideklarasikan private sehingga variabel p2 (yang dideklarasikan sebagai kelas Parent) tidak dapat mengaksesnya. Kode di atas melanggar aturan dalam semantik program Java.

1.6 *Polymorphism*

Polymorhpisme berarti kemampuan untuk mengambil beberapa bentuk. Dalam konsep pemrograman, hal tersebut berarti kemampuan dari sebuah variabel dengan tipe tertentu untuk mengacu ke tipe obyek yang berbeda.

Dalam contoh tentang kelas Manager dan Employee di atas, dapat dibuat obyek Manager dan diisikan ke variabel bertipe Employee. Hal ini dapat dilakukan. Kelas Manager mewarisi semua atribut dan *method* dari kelas induknya yang juga berarti bahwa setiap operasi yang dilakukan pada obyek Employee juga dapat dilakukan pada obyek Manager.

```
Employee employee = new Manager();
employee.name = “James”;
employee.department = “marketing”;    //illegal
```

Dalam contoh di atas, dengan menggunakan variabel employee ini, dapat diakses bagian obyek yang merupakan bagian dari kelas Employee (employee.name), tetapi bagian obyek yang merupakan bagian dari kelas Manager (employee.department) tidak dapat diakses.

1.6.1 *Virtual Method Invocation*

Dalam contoh lain, misalkan dibuat kode sebagai berikut :

```
Employee e = new Employee();
Manager m = new Manager();
```

Jika dipanggil method e.getDetails() dan m.getDetails(),akan diperoleh perilaku yang berbeda. Pemanggilan method yang pertama akan memanggil method getDetails versi kelas Employee sementara pemanggilan method yang kedua akan memanggil method getDetails versi kelas Manager. Tetapi, bagaimana jika kasus berikut ditemukan :

```
Employee e = new Manager();
e.getDetails();
```

Mirip dengan penggalan kode di atas, dibuat obyek Manager dan meng-mengisikannya ke dalam variabel referensi bertipe Employee. Lalu, dipanggil method getDetails() melalui variabel referensi ini. Method getDetails() manakah yang akan dipanggil pada saat *runtime*, method getDetails() versi kelas Manager atau Employee ? Jawabannya adalah method getDetails() versi kelas Manager. Hal ini berhubungan dengan konsep yang disebut *virtual method invocation*. Dalam konsep ini, method mana yang akan dipanggil pada saat *runtime* akan ditentukan berdasarkan tipe aktual obyeknya. Dalam contoh di atas, karena yang dimiliki adalah obyek yang bertipe Manager, maka pemanggilan e.getDetails() akan memanggil method getDetails() versi kelas Manager.

1.6.2 Argumen *Polymorphism*

Method yang menerima sebuah obyek “generik” (dalam hal ini kelas Employee) dapat didefinisikan dan method tersebut akan berperilaku sesuai dengan tipe obyek aktualnya. Sebagai contoh, dibuat sebuah *method* yang mengambil argumen obyek Employee lalu menggunakan argumen tersebut untuk melakukan perhitungan terhadap jumlah pajak

upah (*salary tax*) yang harus dibayarkannya. Hal tersebut dapat dibuat dengan menggunakan cara *polymorphism* sebagai berikut :

```
//dalam class Employee
public TaxRate findTaxRate(Employee e) {
   //perhitungan pajak dan pengembalian hasil perhitungan
}

//di suatu tempat lain
Manager m = new Manager();
TaxRate t = findTaxRate(m);
```

Hal ini dapat dilakukan karena seorang `Manager` adalah seorang `Employee` (karyawan).

1.7 *Casting Object*

Sering dijumpai keadaan di mana suatu *method* menerima variabel referensi dari tipe kelas induk. Lalu digunakan operator `instanceof` untuk memastikan bahwa obyek yang dimiliki adalah tipe *subclass* nya. Setelah melakukan hal tersebut, dapat dikembalikan semua fungsionalitas obyek *subclass* tersebut dengan meng-*casting* variabelnya. Contoh berikut menggambarkan konsep *casting* obyek :

```
public void doSomething(Employee e) {
   if(e instanceof Manager) {
      Manager m = (Manager)e;
      System.out.println("this is the manager of " +
                         m.getDepartment());
   }
   //rest of operation
}
```

Dalam contoh di atas, *method* menerima variabel referensi (e) bertipe `Employee`. Selanjutnya digunakan operator `instanceof` untuk memastikan bahwa obyek yang dimiliki tersebut merupakan *instance* dari kelas `Manager`. Jika hal itu benar, variabel referensi tersebut di-*casting* ke variabel referensi yang bertipe `Manager` (`m`). Dengan menggunakan variabel referensi yang bertipe `Manager` ini, dapat dipanggil *method* yang terdapat di kelas `Manager`, tetapi tidak ada di kelas `Employee`, yaitu method `getDepartment()`. Jika tidak menggunakan *casting*, maka usaha untuk memanggil *method* `e.getDepartment()` pada baris ke-3 akan gagal karena *compiler* tidak dapat menemukan method `getDepartment()` di dalam kelas `Employee`.

1.8 Memanggil Konstruktor *Parent Class*

Seperti halnya *method*, konstruktor dapat memanggil konstruktor kelas induk yang berada langsung di atasnya yang tidak dideklarasikan `private`. Sering didefinisikan konstruktor yang menggunakan beberapa argumen dan menggunakan argumen tersebut untuk mengontrol pembuatan bagian induk dari sebuah obyek. Satu jenis konstruktor induk dapat dipanggil sebagai bagian dari inisalisasi kelas anaknya. Caranya adalah dengan menggunakan *keyword* `super` pada baris pertama dari konstruktor anaknya. Untuk memanggil konstruktor tertentu dari kelas induk, harus disediakan argumen yang sesuai pada pemanggilan `super()`.

Ketika pemanggilan konstruktor induk dengan argumen tidak ditemui, maka yang akan dipanggil secara implisit adalah konstruktor induk yang tidak memiliki argumen. Jika ternyata di kelas induk tidak ditemui konstruktor tanpa argumen, hal tersebut akan menyebabkan kesalahan *compiler*.

Pemanggilan konstruktor induk dengan cara memanggil `super()` ini dapat menggunakan beberapa argumen, yang disesuaikan dengan argumen yang ada di dalam konstruktor-konstruktor di kelas induknya. Hal yang patut diperhatikan adalah pemanggilan `super()` ini harus berada di baris pertama dari konstruktor anaknya.

Jika kelas `Employee` mempunyai set konstruktor sebagai berikut :

```
public class Employee {
   private static final double BASE_SALARY = 15000.00
   private String name;
   private double salary;
   private Date birthDate;

   public Employee(String name, double salary, Date DoB) {
      this.name = name;
      this.salary = salary;
      this.birthDate = DoB;
   }

   public Employee(String name, double salary) {
      this(name, salary, null);
   }

   public Employee(String name, Date DoB) {
      this(name, BASE_SALARY, DoB);
   }

   public Employee(String name) {
      this(name, BASE_SALARY);
   }

}
```

maka dapat dipanggil set konstruktor tersebut dari kelas Manager yang meng-*extends* kelas Employee dengan cara berikut :

```
public class Manager extends Employee {
   private String department;

   public Manager(String name, double salary, String dept){
      super(name, salary);
      department = dept;
   }
   public Manager(String name, String dept) {
      super(name);
      department = dept;
   }
   public Manager(String dept) {
      department = dept;
   }
}
```

Konstruktor pada baris 12 menyebabkan kesalahan *compile* karena *compiler* menyisipkan panggilan super() secara implisit., dan tidak ada konstruktor di kelas induknya (Employee) yang tidak memiliki argumen. Pernyataan super atau this harus dituliskan pada baris pertama dari konstruktor. Jika tidak menuliskan super(…) atau this(…) pada baris pertama konstruktor, maka *compiler* akan menyisipkan secara otomatis panggilan terhadap super() tanpa menggunakan argumen.

Konstruktor lainnya dapat pula memanggil super() atau this(), di mana hal ini akan mengakibatkan pemanggilan konstruktor secara berantai (*constructor chaining*). Hal yang patut diperhatikan adalah konstruktor pada kelas induk akan lebih dulu dieksekusi sebelum konstruktor pada kelas anaknya.

2 ARRAY

2.1 Mendeklarasikan Array

Array biasanya digunakan untuk mengelompokkan data atau obyek yang memiliki tipe yang sama. Array memungkinkan untuk mengacu ke sekumpulan obyek dengan nama yang sama.

Array dapat dideklarasikan dengan tipe apa saja, baik itu yang bertipe data primitif maupun obyek. Berikut contoh deklarasi Array :

```
Char s[];
Point p[];    // Point adalah sebuah kelas
```

Dalam bahasa pemrograman Java, Array merupakan sebuah obyek meskipun ia terdiri dari elemen yang bertipe data primitif. Seperti halnya kelas yang lain, ketika mendeklarasikan Array belum dibentuk sebuah obyek Array. Deklarasi Array hanya membuat sebuah referensi yang dapat digunakan untuk mengacu ke sebuah obyek Array. Besarnya memori yang digunakan oleh elemen-elemen Array akan dialokasikan secara dinamis dengan menggunakan pernyataan new atau dengan *array initializer*.

Contoh deklarasi array di atas, dengan menggunakan kurung siku setelah nama variabel, merupakan standar penulisan array dalam C, C++ dan Java. Format demikian agak sulit untuk dibaca. Oleh karenanya, bahasa Java menggunakan bentuk deklarasi array alternatif dengan menggunakan kurung siku di sebelah kiri seperti dalam contoh berikut :

```
Char [] s;
Point [] p;    // Point adalah sebuah kelas
```

Sebuah deklarasi array dapat ditulis sebagai pendeklarasian tipe array di bagian kiri dan nama variabel di bagian kanan. Kedua bentuk deklarasi array di atas dapat digunakan, tetapi sebaiknya konsisten dalam menggunakan satu bentuk saja. Dalam deklarasi array, tidak ditentukan ukuran aktual dari array.

Ketika mendeklarasikan array dengan kurung siku yang berada di sebelah kiri nama variabel, kurung siku tersebut berlaku bagi semua variabel yang berada di sebelah kanannya.

2.2 Membuat Array

Seperti halnya dalam pembuatan obyek, Array dibuat dengan menggunakan keyword new. Contoh di bawah ini membuat sebuah Array dengan tipe primitif char:

```
s = new char[26];
```

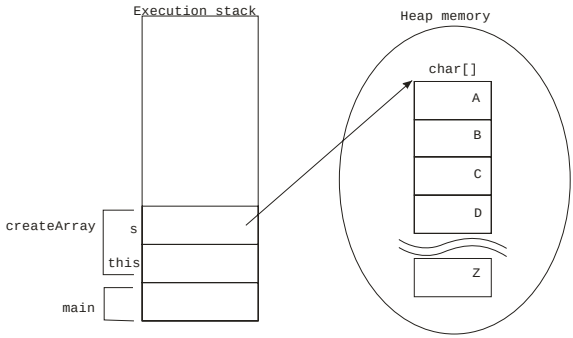
Baris di atas membuat sebuah array dengan 26 nilai char. Ketika membuat sebuah Array, elemen di dalamnya akan diinisialisasi dengan nilai default (‘\u0000’ untuk data tipe char). Biasanya nilai elemen Array dapat diinisialisasi dengan nilai *default* sebagaimana ditunjukkan berikut ini :

```
s[0] = ‘A’;
s[1] = ‘B’;
...
```

Perhatikan penggalan kode sebagai berikut :

```
public char[] createArray {
        char[] s;

        s = new char[26];
        for(int i=0; i<26; i++) {
                s[i] = (char) ('A' + i);
        }
        return s;
}
```

Penggalan kode di atas mempunyai representasi dalam heap memory sebagai berikut :



Gambar 2.1 Representasi Array primitif dalam *heap memory*.

Indeks array dimulai dengan angka 0. Batasan legal untuk nilai indeks ini adalah dari nol hingga jumlah elemen array – 1. Apabila program berusaha mengakses array di luar batas yang legal, maka akan muncul *runtime exception*.

Untuk membuat Array dengan elemen obyek, gunakan cara penulisan berikut :

```
p = new Point[10];
```

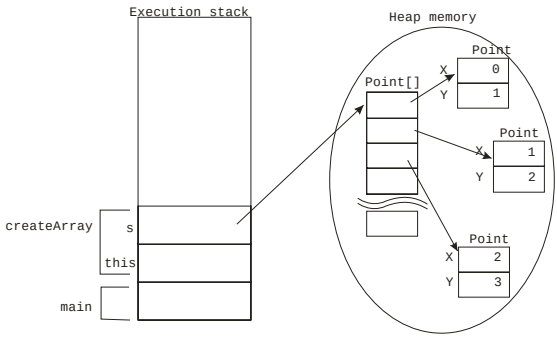
Pada kode di atas, dibuat sebuah Array yang masing-masing elemennya merupakan referensi obyek yang bertipe Point. Pernyataan tersebut tidak membuat 10 obyek Point. Untuk membuat obyek Point, gunakan pernyataan berikut :

```
P[0] = new Point(0, 1);
P[1] = new Point(1, 2);
 ...
```

Perhatikan penggalan kode program berikut ini.

```
public Point[] createArray() {
  Point[] p;
  p = new Point[10];
  for ( int i=0; i<10; i++ ) {
    p[i] = new Point(i, i+1);
  }
  return p;
}
```

Representasi Array p di dalam heap memory adalah sebagai berikut :



Gambar 2.2 Representasi Array bertipe obyek dalam *heap memory*.

2.3 Menginisialisasi Array

Ketika membuat sebuah Array, setiap elemen di dalamnya akan diinisialisasi dengan nilai *default*. Dalam kasus Array char s seperti pada contoh di atas, setiap elemen dari Array s diinisialisasi dengan nilai '\u0000'. Dalam kasus Array p seperti juga contoh di atas, setiap elemen diinisialisasi dengan nilai null, yang berarti elemen-elemen array p tersebut belum merujuk ke suatu obyek Point. Setelah proses pengisian p[0] = new Point(), barulah diperoleh elemen pertama dari Array p yang merujuk ke obyek Point.

Java juga memiliki cara singkat untuk membuat sebuah obyek Array dengan elemen-elemennya memiliki nilai awal :

```
String names[] = {"Georgianna","Jen","Simon"};
```

Kode di atas adalah sama dengan kode berikut ini:

```
String names[];
names[0] = "Georgianna";
names[1] = "Jen";
names[2] = "Simon";
```

Cara singkat ini dapat digunakan untuk Array dengan berbagai jenis tipe, seperti dalam contoh berikut :

```
MyDate dates[] = {
        new MyDate(22, 7, 1964),
        new MyDate(1, 1, 2000),
        new MyDate(22, 12, 1964),
};
```

2. 4 Array Multidimensi

Dalam Java, dapat dibuat Array dari Array sehingga dinamai Array multidimensi. Contoh berikut ini memperlihatkan cara membuat Array dua dimensi :

```
int twoDim [][] = new int[4][];
twoDim[0] = new int[5];
twoDim[1] = new int[5];
```

Pada baris pertama, dibuat sebuah obyek Array yang memiliki empat elemen. Masing-masing elemen merupakan referensi yang bertipe Array dari int yang bernilai null. Pada baris berikutnya, diinisialisasi elemen pertama dan kedua dengan obyek Array dari int yang masing-masing berisi lima elemen yang bertipe *integer*.

Array dari Array yang bersifat *non-rectangular* dapat dibuat seperti dalam contoh berikut :

```
twoDim[0] = new int[2];
twoDim[1] = new int[4];
twoDim[2] = new int[6];
twoDim[3] = new int[8];
```

Proses pembuatan Array dari Array yang demikian, di mana harus diinsialisasi masing-masing elemennya satu persatu, cukup merepotkan. Di samping itu, array dari array yang berbentuk *rectangular* lebih sering digunakan dibandingkan dengan yang non-*rectangular*. Untuk membuat Array dari Array yang *rectangular* dengan cara mudah, gunakan bentuk berikut ini :

```
int twoDim[][] = new int[4][5];
```

Kode di atas membuat sebuah Array dari empat Array yang masing-masing elemennya berisi lima nilai int.

2.5 Batasan Array

Dalam bahasa Java, index Array dimulai dengan angka nol. Jumlah elemen di dalam Array disimpan sebagai bagian dari obyek Array di dalam atribut length. Atribut ini dapat digunakan untuk melakukan proses iterasi pada Array seperti dalam contoh berikut :

```
int list[] = new int[10];
for(int i = 0; i < list.length; i++) {
        System.out.println(list[i]);
}
```

Dengan menggunakan atribut length, pemeliharaan kode program menjadi mudah karena program tidak perlu mengetahui jumlah eleman Array pada saat kompilasi.

2. 6 Manipulasi Array

2.6.1 Mengubah Ukuran Array

Setelah membuat obyek Array, ukuran Array tersebut tidak dapat diubah. namun demikian, dapat digunakan variabel referensi yang sama untuk menunjuk ke sebuah obyek Array baru seperti dalam contoh di bawah ini :

```
int myArray[] = new int[6];
myArray = new int[10];
```

Dalam kode program ini, obyek Array yang pertama akan hilang kecuali ada variabel lain yang dibuat untuk merujuk ke obyek Array tersebut.

2.6.2 Menyalin Array

Bahasa Java menyediakan *method* khusus untuk menyalin Array, yaitu dengan menggunakan method arrayCopy() dari kelas System seperti dalam contoh berikut :

```
// array semula
int myArray[] = {1, 2, 3, 4, 5, 6};

// array engan elemen yang lebih banyak
int hold[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

//menyalin semua elemen myArray ke hold
// dimulai dengan indeks ke 0
System.arraycopy(myArray, 0, hold, 0,myArray.length);
```

Setelah proses pengkopian di atas, maka isi dari Array hold adalah 1, 2, 3, 4, 5, 6, 4, 3, 2, 1.

3 PENGGUNAAN KELAS DI JAVA

3.1 Atribut Kelas

Pada saat tertentu, terkadang dibutuhkan variabel yang di-*share* oleh semua *instance* kelas. Sebagai contoh, variabel yang digunakkan sebagai basis untuk komunikasi antar *instance* atau menghitung jumlah *instance* yang telah dibuat. Untuk melakukan hal ini, tandai sebuah variabel dengan *keyword* `static`. Variabel tersebut seringkali disebut sebagai variabel kelas untuk membedakannya dari *member variable* atau variabel *instance* yang tidak di-*share* oleh semua *instance*. Perhatikan contoh kode program berikut.

```
public class Count1 {
  private int serialNumber;
  private static int counter = 0;

  public Count1() {
        counter++;
        serialNumber = counter;
  }
}
```

Dalam contoh di atas, diberikan *serial number* yang unik untuk setiap obyek yang dibuat, dimulai dengan angka 1 untuk obyek pertama, 2 untuk obyek kedua dan seterusnya. Variable `counter` di-*share* oleh semua *instance*. Ketika membuat sebuah obyek `Count`, konstruktor akan dipanggil. Pemanggilan konstruktor ini akan meng-*increment* nilai dari variabel `counter` dan nilai ini diberikan kepada variabel `serialNumber`.

Variabel `static` memiliki konsep yang serupa dengan variabel global di dalam bahasa pemrograman lainnya. Bahasa Java tidak mengenal variabel global, tetapi variabel `static` adalah sebuah variabel yang dapat diakses oleh setiap *instance* dari sebuah kelas. Jika variabel `static` di atas (`counter`) tidak dideklarasikan `private`, maka variabel tersebut dapat diakses dari luar kelas. Untuk mengkasesnya tidak diperlukan *instance* dari kelasnya, tetapi menggunakan nama kelasnya sebagaimana ditunjukkan pada kode program berikut ini.

```
public class OtherClass {
  public void incrementNumber() {
  Count1.counter++;
  }
}
```

3.2 *Method* Statik

Terkadang perlu juga mengakses kode program dimana *instance* dari sebuah kelas tidak dimiliki. Hal tersebut dapat dilakukan dengan menggunakan *method* yang dideklarasikan `static`. *method* demikian seringkali disebut sebagai *method* kelas sebagaimana ditunjukkan pada kode program berikut.

```
public class Count2 {
  private int serialNumber;
  private static int counter = 0;

  public static int getTotalCount() {
    return counter;
  }

  public Count2() {
    counter++;
    serialNumber = counter;
  }
}
```

Untuk menggunakan *method* yang dideklarasikan `static`, tidak digunakan referensi obyeknya melainkan nama kelas seperti ditunjukkan pada kode program berikut ini :

```
public class TestCounter {
  public static void main(String[] args) {
    System.out.println("number of counter is "+
    Count2.getTotalCount());
    Count2 count1 = new Count2();
    System.out.println("number of counter is "+
    Count2.getTotalCount());
  }
}
```

Keluaran dari program `TestCounter` di atas adalah :

```
Number of counter is 0
Number of counter is 1
```

Karena *method* `static` dipanggil tanpa menggunakan variabel *instance* dari kelas yang bersangkutan, maka tidak dapat diterapkan variabel referensi `this`. Akibatnya, *method* `static` tidak dapat mengakses

variabel selain daripada variabel lokal, atribut `static`, dan parameternya. Apabila berusaha mengakses atribut non-`static` dari *method* `static`, hal itu akan menyebabkan *compiler error*.

Atribut *non-static* terkait dengan satu *instance* tertentu dan ia hanya bisa diakses melalui referensi *instance*.

```
public class Count3 {
  private int serialNumber;
  private static int counter = 0;

  public static int getSerialNumber() {
        return serialNumber; //COMPILER ERROR
  }
}
```

Hal yang harus diperhatikan ketika menggunakan *method* `static` antara lain :

- *Method* `static` tidak dapat di-*override*, tetapi dapat disembunyikan.
- Method `main()` merupakan sebuah method `static` karena JVM tidak perlu membuat sebuah *instance* dari kelas yang bersangkutan ketika method `main()` dieksekusi. Jika mempunyai *data member*, hendaknya dibuatkan sebuah obyek untuk mengaksesnya.

3.3 *Keyword* final

Keywod `final` dapat digunakan untuk deklarasi kelas, *method*, dan variabel.

3.3.1 Kelas final

Bahasa pemrograman Java memungkinkan penggunaan *keyword* `final` pada kelas. Jika kelas dideklarasikan final, maka kelas tersebut tidak dapat diturunkan. Salah satu contoh dari kelas `final` adalah kelas `java.lang.String`. Kelas ini dibuat `final` untuk alasan keamanan.

3.3.2 *Method* final

Method dapat dideklarasikan dengan *keyword* `final`. Method yang dideklarasikan `final` tidak dapat di-*override*. Sebuah *method* sebaiknya didelkarasikan `final` jika *method* yang bersangkutan mempunyai implementasi yang tidak boleh diubah dan penting untuk menjaga konsistensi perilaku sebuah obyek. Selain itu, method `final` juga digunakan karena alasan optimisasi. Dengan *method* `final`, *compiler* dapat membuat kode yang memanggil langsung *method* yang

bersangkutan, tidak menggunakan *virtual invocation method* yang melibatkan *runtime lookup*.

Method yang dideklarasikan `static` dan `private` juga dioptimasi oleh *compiler* seperti halnya *method* yang dideklarasikan `final`.

3.3.3 Variabel final

Jika variabel dideklarasikan dengan *keyword* `final`, maka variabel tersebut dijadikan sebagai sebuah konstanta. Apabila program berusaha mengubah nilai dari variabel `final`, maka akan muncul kesalahan *compiler*. Kode program berikut ini menggambarkan penggunaan variabel `final` :

```
public class Bank {
  private static final double DEFAULT_INTEREST_RATE=3.2;
  ...
}
```

Untuk kasus variabel referensi, jika menandai variabel tersebut dengan `final`, maka variabel tersebut tidak dapat merujuk ke obyek lainnya. Tetapi, isi variabel dari obyek tersebut dapat diubah.

Blank final variable adalah sebuah variabel `final` yang tidak diinisialisasi ketika dideklarasikan. Biasanya *Blank final variable* yang merupakan variabel *instance* diberikan nilainya di dalam konstruktor. Variabel kosong final yang merupakan variabel lokal dapat diset nilainya di bagian manapun di dalam *method*, tetapi hanya dapat diset satu kali. Kode program berikut menunjukkan penggunaan variabel final dengan nilai awal kosong.

```
public class Customer {
  private final long customerID;

  public Customer() {
    customerID = createID();
  }

  public long getID() {
    return customerID;
  }

  private long createID() {
    return ...// generate new ID
  }

  //more declarations
}
```

3.4 *Keyword* abstract

Keyword abstract dapat diterapkan pada kelas dan *method*.

3.4.1 Kelas abstrak

Sebuah kelas dapat dideklarasikan sebagai kelas abstrak. Tujuan membuat kelas abstrak adalah agar satu kelas lain dapat memperluasnya (extend) dengan jalan menjadi *subclass* darinya. Sebagai contoh, bayangkan seekor hewan yang mempunyai *method* generik yang sama, yang dimiliki oleh semua hewan. Keadaan spesifik dari hewan tersebut seperti warnanya, jumlah kakinya, suaranya, dan sebagainya belum diiketahui. Untuk itulah, diperlukan kelas abstrak. Kelas tersebut tidak dapat diinstansiasi. Hanya turunan dari kelas abstrak yang dapat diinstansiasi. Berikut ini contoh kelas abstrak:

```
public abstract class Animal {
  private int numOfFoots;
  private Color color;

  public abstract void walk();
  public abstract void sound();
}


public class Cat extends Animal {
  public void walk() {
    System.out.println("cat walks using 4 legs");
  }
  public void sound() {
    System.out.println("miaouw");
  }
}

public class Chicken extends Animal {
  public void walk() {
    System.out.println("chicken walks using 2 legs");
  }
  public void sound() {
    System.out.println("kukuruyuk");
  }
}
```

Pada kode program di atas dideklarasikan kelas Animal yang abstrak, di mana di dalamnya dideklarasikan dua variabel yaitu numOfFoots dan color. Selain itu, dapat juga dideklarasikan dua buah *method*, yaitu walk(), dan sound(). Dalam *method* di dalam kelas Animal, tidak didefiniskan impelementasi *method* yang dideklarasikannya, dengan harapan bahwa obyek turunan hewan yang spesifik, seperti Chicken dan Cat akan menyediakan implementasinya yang spesifik dengan

kondisi masing-masing hewan. Dalam contoh di atas, method abstrak `walk()` dan `sound()` dalam kelas `Animal` diimplementasikan oleh kelas `Chicken` dan `Cat`.

Kelas abstrak dapat terdiri dari satu atau lebih *method* abstrak seperti contoh di atas. Kelas abstrak juga boleh tidak memiliki method abstrak sebagaimana ditunjukkan pada program berikut.

```
public abstract class Vehicle {
  private String type;

  public void getType() {
    return type;
  }
}
```

Kelas `Vehicle` di atas merupakan contoh kelas abstrak yang legal. Apabila ada satu saja *method* yang dideklarasikan dalam sebuah kelas sebagai *method* abstrak, maka kelas tersebut harus dideklarasikan `abstract`.

3.4.2 *Method* abstrak

Method abstrak adalah *method* yang sudah dideklarasikan, tetapi belum diimplementasikan. Sebuah *method* ditulis dengan *keyword* `abstract` ketika kelas turunan (*subclass*) di bawahnya menyediakan implementasi *method* tersebut.

Method abstrak ditandai dengan *modifier* `abstract`, tidak memiliki implementasi, dan diakhiri dengan tanda titik koma (;).

```
public void myAbstractMethod();
```

3.5 *Interface*

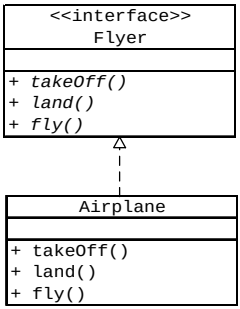
Interface mendefinisikan aturan perilaku (*protocol of behavior*) yang dapat diimplementasikan oleh kelas manapun. *Interface* mendefinisikan satu set *method* tanpa menyediakan implementasinya. Sebuah kelas yang mengimplementasi sebuah *interface* terikat kontrak oleh *interface* tersebut untuk mengimplementasi semua *method* yang ada di dalam *interface*. Dengan kata lain, kelas tersebut terikat untuk meng-implementasikan perilaku tertentu yang tertulis dalam *interface*. Secara substansi, *interface* merupakan kumpulan dari *method* abstrak dan konstanta.

Interface memiliki kemiripan dengan kelas abstrak, di mana keduanya memuat *method* abstrak. Perbedaan penting antara *interface* dan kelas abstrak adalah :

- Sebuah *interface* tidak dapat membuat implementasi satu *method* pun, sementara kelas abstrak dapat membuat implementasi satu atau lebih method abstrak
- Sebuah kelas dapat mengimplementasi beberapa *interface*, tetapi ia hanya dapat meng-*extends* satu *superclass*.
- *Interface* bukan merupakan bagian dari hirarki kelas. Dua kelas yang tidak berhubungan dalam jalur hirarki kelas dapat mengimplementasi *interface* yang sama.

3.5.1 Deklarasi *interface*

Misalkan ada sekumpulan obyek yang memiliki kemampuan yang sama, yaitu mereka semua dapat terbang. Untuk itu dapat dibuatkan sebuah *interface* publik, yang dinamakan `Flyer`, yang memiliki tiga operasi `takeOff`, `land`, dan `fly`.



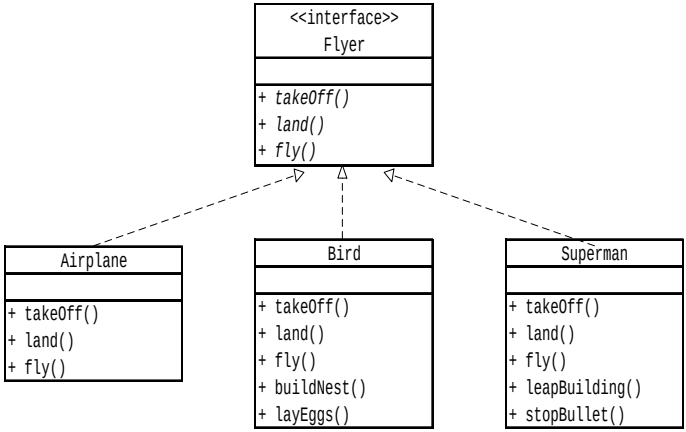
Gambar 3.1 *Interface* `Flyer` dan implementasinya oleh kelas `Airplane`.

Panah dengan garis putus-putus di atas menggambarkan kelas `Airplane` mengimplementasikan *interface* `Flyer`. Kode Java dari disain kelas di atas ditunjukkan sebagai berikut.

```
public interface Flyer {
  public void takeOff();
  public void land();
  public void fly();
}
```

```
public class Airplane implements Flyer {
  public void takeOff() {
    //accelerate until lift-off
    //raise landing gear
  }

  public void land() {
    //lower landing gear
    //decelerate and lower flaps until touch-down
    //apply breaks
  }

  public void fly() {
    //keep those engines running
  }
}
```

Satu *interface* `Flyer` dapat diimplementasikan oleh beberapa kelas. Sebagai contoh, pesawat terbang (`Airplane`), burung (`Bird`), dan superman (`Superman`), semuanya dapat terbang.

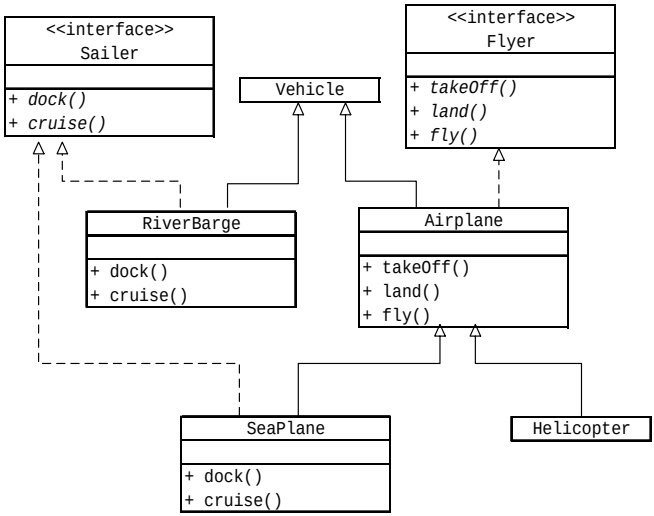


Gambar 3.2 Implementasi *interface* `Flyer` oleh beberapa kelas.

Sebuah kelas `Airplane` mungkin merupakan *subclass* dari kelas `Vehicle` (kendaraan), dan ia dapat terbang. Kelas `Bird` merupakan *subclass* dari kelas `Animal` (hewan), dan ia pun dapat terbang. Contoh di atas menggambarkan bagaimana sebuah kelas dapat mewarisi satu kelas tetapi mengimplementasi beberapa *interface* sekaligus.

3.5.2 Implementasi *multiple interface*

Seperti telah dijelaskan di atas, sebuah kelas dapat mengimplementasi lebih dari satu *interface*. Sebagai contoh, sebuah *SeaPlane* (pesawat yang dapat mendarat dan terbang di/dari air) tidak hanya dapat terbang, ia juga dapat berlayar. Kelas `SeaPlane` meng-*extends* kelas `Airplane`, yang berarti ia juga mewarisi implementasi dari *interface* `Flyer`. Kelas `SeaPlane` juga mengimplementasi *interface* `Sailer`.



Gambar 3.3 Contoh dari konsep implementasi *multipe interface*.

3.5.3 Penggunaan *interface*

Biasanya *interface* digunakan untuk:

- Mendeklarasikan *method* yang akan diimplementasikan oleh satu atau beberapa kelas.
- Menunjukkan *interface* sebuah obyek ke publik tanpa menunjukkan isi kelas sebenarnya.
- Menangkap kesamaan di antara beberapa kelas tanpa perlu memasukkannya dalam hirarki kelas (*superclass* - *subclass*).
- Mensimulasikan konsep pewarisan banyak kelas dengan mendeklarasikan kelas yang mengimplementasikan beberapa interface sekaligus.

3.6 Kelas bersarang (*nested class*) dan *inner class*

Kelas bersarang merupakan sebuah kelas yang dideklarasikan di dalam kelas lainnya. Berikut merupakan contoh dari sebuah kelas bersarang.

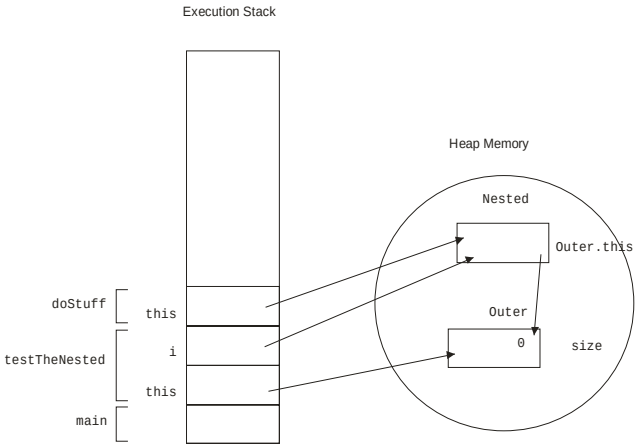
```
public class EnclosingClass {
  //more class definition here

  public class ANestedClass {
    //more class definition here
  }
}
```

Kelas bersarang digunakan untuk membuat hubungan antara dua kelas. Ketika mendefinisikan sebuah kelas di dalam kelas lainnya, hendaknya dipikirikan bahwa keberadaan kelas bersarang tersebut hanya akan bermakna ketika ia berada di dalam konteks kelas pembungkusnya (*enclosing class*). Atau dalam bahasa lain, untuk memfungsikannya, kelas bersarang tersebut bergantung pada *enclosing class*-nya.

Sebagai anggota dari *enclosing class*, kelas bersarang memiliki satu kelebihan. Ia dapat mengakses semua anggota dari *enclosing class*, termasuk anggota yang dideklarasikan `private`.

```
public class Outer1 {
private int size;

  /*declare a nested class called "Nested"*/
  public class Nested {
    public void doStuff() {
      //the nested class has access to 'size' from Outer
      size++;
    }
  }

  public void testTheNested() {
    Nested i = new Nested();
    i.doStuff();
  }
}
```

Dalam contoh di atas, kelas `Outer1` mendeklarasikan atribut bernama `size`, sebuah kelas bersarang yang bernama `Nested`, dan sebuah *method* yang bernama `testTheNested`. Di dalam kelas `Nested`, dideklarasikan sebuah *method* yang bernama `doStuff`. *Method* ini mempunyai akses ke kelas `Outer1`, di mana variabel `size` di dalam *method* tersebut (baris 8) mengacu ke data atribut `size` dalam kelas `Outer1` (baris 2). Hal ini berarti bahwa obyek bersarang (*nested*) mempunyai referensi ke obyek pembungkusnya (*outer*). Gambar berikut menunjukkan bagaimana hal ini diimplementasikan dalam JVM.

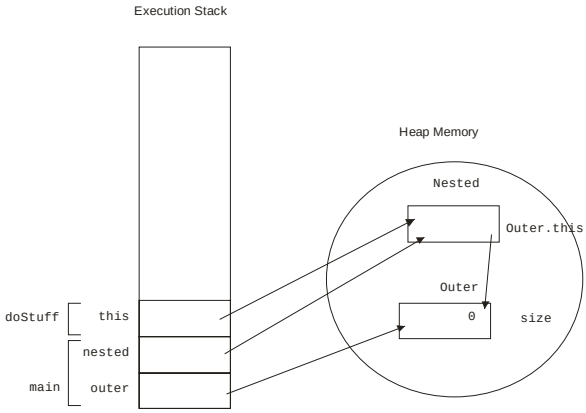


Gambar 3.4 Representasi dari kelas bersarang di dalam memori.

Contoh berikut ini menggambarkan bagaimana menginstantiasi obyek dari kelas bersarang yang dideklarasikan public dari luar *enclosing class*.

```
public class Outer2 {
  private int size;

  public class Nested {
    public void doStuff() {
    size++;
  }
}
}

public class TestNested {
  public static void main(String[] args) {
    Outer2 outer = new Outer2();

    //must create a Nested object relative to an Outer
    Outer2.Nested nested = outer.new Nested();
    nested.doStuff();
  }
}
```

Seperti dalam penggalan kode di atas, obyek kelas `Nested` diinstansiasi dalam konteks *instance enclosing class* nya, dalam hal ini kelas `Outer2` (baris 6 dari method main kelas `TestNested`).

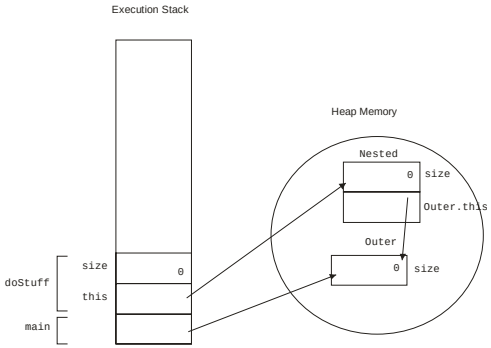


Gambar 3.5 Representasi dari kelas bersarang di dalam memori.

Contoh berikut ini menggambarkan identifikasi variabel dengan nama yang sama.

```
public class Outer3 {
   private int size;

   public class Nested {
      private int size;

      public void doStuff(int size) {
         size++;          //the local parameter
         this.size++; //the Nested object attribute
         Outer3.this.size++; //the Outer3 object attribute
      }
   }
}
```

Variabel size digunakan dalam tiga konteks, yaitu: sebagai atribut data dari kelas `Outer3`, sebagai atribut data dari kelas `Nested`, dan sebagai variabel lokal dari method `doStuff`. Hal demikian diperbolehkan dalam bahasa Java, tetapi tidak dianjurkan dalam pemrograman.



Gambar 3.6 Mengidentifikasi variabel dalam konteks yang berbeda.

Dalam contoh berikut, dibuatkan sebuah kelas bersarang dalam lingkup sebuah *method*.

```
public class Outer4 {
   private int size = 5;

   public Object makeTheNested(int localVar) {
      final int finalLocalVar = 6;

      //declare a class within a method
      class Nested {
         public String toString() {
            return ("nested size=" + size +
            //"localVar=" + localVar + //ERROR: ILLEGAL
            "finalLocalVar=" + finalLocalVar);
         }
      }

      return new Nested();
   }


   public static void main(String[] args) {
      Outer4 outer = new Outer4();
      Object obj = outer.makeTheNested(47);
      System.out.println("the object is "+ obj);
   }
}
```

Kelas yang dideklarasikan di dalam *method* seperti contoh di atas dinamakan *local class*. Konsep penting dari *local class* adalah *method-method* di dalam *local class* tidak mempunyai akses penuh di dalam *method* yang melingkupinya (*outer method*). Sebagai contoh, misalkan suatu *method* membuat obyek *nested class* dan mengembalikan obyek

tersebut (baris 16). Setelah keluar dari method `makeTheNested`, variabel lokal yang ada di dalam method `makeTheNested` pun hilang, sehingga method di dalam *nested class* (seperti method `doStuff`) tidak dapat mengakses variabel tersebut pada saat runtime. Oleh karenanya, pesan *error compiler* akan muncul apabila baris 11 dijalankan. Tetapi, *local class* dapat menggunakan variabel lokal yang dideklarasikan `final` karena variabel yang demikian tetap ada meskipun suatu *method* telah dikembalikan.

Kelas bersarang memiliki bebeapa sifat, antara lain :

- Kelas bersarang dapat dirujuk dengan menggunakan nama nya saja (`Nested`) dari dalam lingkup *enclosing class*. Di luar lingkup tersebut, harus digunakan nama lengkapnya (`Outer.Nested`). Nama dari kelas bersarang harus berbeda dengan nama *enclosing class*-nya.
- Kelas di dalam *method* dapat didefinisikan. Kelas yang demikian dinamakan *local class*.
- *Local class* dapat mengakses variabel lokal, termasuk argumen dalam *method* dari *method* yang melingkupinya, dengan syarat variabel tersebut dideklarasikan `final`.
- Kelas bersarang dapat dideklarasikan `abstract`.
- Sebuah *interface* dapat dideklarasikan di dalam *interface* lainnya (*nested*). *Interface* yang demikian dapat diimplementasi secara biasa oleh sebuah kelas atau oleh kelas bersarang.
- Sebuah kelas bersarang dapat mengakses member `static` dari *enclosing class*.
- A*ccess control* apa pun dapat digunakan untuk sebuah kelas bersarang (`public`, `protected`, *default*, dan `private`), kecuali *local class* (kelas yang dideklarasikan di dalam *method*). Sebagai contoh, kelas bersarang yang dideklarasikan `private` hanya dapat diakses di dalam lingkup *enclosing class*, kelas bersarang yang dideklarasikan `protected` dapat digunakan oleh *subclass* dan seterusnya. Pemberian access control pada kelas bersarang tidak menghalanginya untuk mengakses atribut pada lingkup *enclosing class*.
- Ketika sebuah file java yang memuat *enclosing class* dan kelas bersarang dikompilasi, maka baik *enclosing class* maupun kelas bersarang tersebut turut dikompilasi. Nama dari kelas bersarang yang telah terkompilasi adalah OuterClass$NestedClass dengan ekstensi .class. Sebagai contoh, kelas bersarang Nested akan menghasilkan Outer$Nested.class pada saat kompilasi.

4 PENANGANAN *EXCEPTION*

4.1 Pengantar *Exception*

Exception merupakan sebuah mekanisme yang digunakan oleh bahasa pemrograman komputer untuk mendeskripsikan apa yang harus dilakukan ketika sesuatu yang tidak dikehendaki terjadi. Sesuatu yang tidak dikehendaki ini biasanya dalam bentuk *error*, seperti pemanggilan *method* dengan arguman yang tidak *valid*, kegagalan koneksi dalam sebuah jaringan, atau usaha user membuka sebuah file yang tidak ada.

Dalam bahasa pemrograman Java, terdapat dua macam *exception*, yaitu *checked exception* dan *unchecked exception*. *Checked exception* adalah *exception* yang muncul akibat dari kondisi external yang dapat saja muncul dari program. *Exception* jenis ini harus ditangani oleh programmer. Contoh dari *checked exception* misalkan kondisi ketika sebuah file yang diminta tidak dapat ditemukan atau adanya *network failure*. *Unchecked exception* merupakan kategori *exception* yang muncul akibat dari adanya *bugs* dalam program atau situasi yang terlalu sulit untuk ditangani. Programmer tidak berkewajiban untuk menangani *exception* jenis ini ketika muncul. Salah satu jenis *unchecked exception* adalah runtime *exception*, yaitu kategori *exception* yang biasanya merepresentasikan *bugs* dalam program. Contoh dari *exception* kategori ini adalah ketika berusaha mengakses index sebuah array di luar *range*-nya. Jenis *unchecked exception* yang lain adalah *error*, yaitu kondisi *exception* yang muncul dari hal-hal yang bersifat *environmental issues* yang sangat sulit untuk ditangani. Contohnya adalah ketika program tidak memiliki cukup memori.

Kelas `Exception` merupakan base kelas dari *checked exception*. Ketika menemui *exception* ini, program tidak menghentikan (*terminate*) jalannya program, tetapi perlu penanganan dan program terus berjalan.

Kelas *Error* adalah base kelas yang digunakan untuk *unchecked exception* yang berasal dari *error* yang fatal dari program yang sulit untuk ditangani. Dalam banyak kasus, ketika menemui *exception* ini, jalannya program dihentikan.

Kelas `RuntimeException` merupakan base kelas untuk *unchecked exception* yang dapat muncul akibat dari adanya bugs dalam program. Sama halnya seperti kasus dalam kelas `Error`, ketika program menemui *exception* ini, maka jalannya program dihentikan.

Ketika sebuah *exception* muncul, *method* di mana *exception* terjadi dapat menangani *exception* tersebut atau melemparkannya ke *method* yang memanggilnya untuk memberi pesan terjadinya sebuah *error*. *Method* yang memanggil method lain ini juga memiliki pilihan yang sama menangani *exception* yang terjadi atau melemparkannya ke method yang memanggilnya. Jika *exception* ini terjadi pada suatu Thread, maka Thread tersebut akan dimatikan. Mekanisme seperti ini memberikan pilihan kepada pembuat program untuk menulis sebuah kode blok untuk menangani *exception* pada tempat-tempat yang sesuai di dalam program. Untuk mengetahui tipe *checked exception* apa saja yang dilempar oleh sebuah *method*, lihat dokumentasi pustaka atau API.

4.2 Penanganan *Exception*

Perhatikan contoh kode berikut ini:

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

    String greetings[] = {
    "Hello world!",
    "No, I mean it!",
    "HELLO WORLD!"
    };

    while(i < 4) {
      System.out.println(greetings[i]);
      i++;
    }
  }
}
```

Jika kode di atas tidak menangani *exception*, maka program akan berhenti dengan sebuah pesan kesalahan.

```
java HelloWorld
Hello world!
No, I mean it!
HELLO WORLD!
java.Lang.ArrayIndexOutOfBounds$Exception$:
at HelloWorld.main(HelloWorld.java:12)
```

Bahasa pemrograman Java menyediakan mekanisme untuk mengetahui *exception* apa yang terjadi dan bagaimana menanganinya.

Mekanisme *exception handling* memungkinkan program untuk menangkap (*catch*) *exception*, menanganinya, dan meneruskan jalannya program. Struktur *exception handling* dibuat sedemikian rupa sehingga *error* dalam program tidak menghalangi jalannya alur normal program. Ketika sebuah *error* terjadi, ia ditangani dalam blok kode

yang terpisah yang diasosiasikan dengan kode eksekusi program yang normal. Hal tersebut akan mempermudah pengaturan kode program.

4.2.1 Pernyataan `try` dan `catch`

Untuk menangani suatu *exception*, tempatkan kode di dalam blok `try` dan membuat satu atau lebih blok `catch`, di mana masing-masing blok `catch` diasosiasikan dengan satu jenis *exception* yang mungkin dilemparkan. Pernyataan di dalam blok `catch` akan dijalankan jika *exception* yang terjadi sesuai dengan salah satu *exception* yang ditangkap oleh salah satu blok `catch` dalam program. Dalam sintaksnya, setelah sebuah blok `try`, dapat ditulis beberapa blok `catch`, seperti dalam contoh berikut:

```
try {
   //code that might throw a particular exception
} catch(myExceptionType myExcept) {
   //code to execute if a MyExceptionType exception is thrown
} catch(Exception otherExcept) {
   //code to execute if a general Exception exception is
thrown
}
```

4.2.2 Mekanisme *call stack*

Jika sebuah pernyataan melemparkan sebuah *exception*, dan *exception* ini tidak ditangani di dalam *method* yang melingkupinya, maka *exception* tersebut akan dilemparkan ke *method* yang memanggilnya (*calling method*). Jika *exception* ini juga tidak ditangani di dalam *method* pemanggil, maka *exception* ini akan dilemparkan ke *method* pemanggil yang memanggil *method* ini. Demikian seterusnya. Jika *exception* tidak juga ditangani ketika ia mencapai *method* `main()`, maka program akan berhenti dengan tidak normal. Perhatikan kode program berikut ini.

```
public class MyClass {
  public static void main(String[] args) {
    first();
    second();
  }

  public static void first() {}

  public static void second() {
    //exception happens here
  }
}
```

Dalam sebuah kasus seperti di atas, misalkan *method* main() memanggil sebuah *method* yang bernama first(), dan *method* ini memanggil *method* lain yang bernama second(). Jika *exception* terjadi di dalam *method* second() dan *method* second() tidak menangani *exception* ini, maka ia akan dilempar ke dalam *method* first(). Misalkan di dalam method first() terdapat blok catch() yang menangani *exception* ini, maka *exception* tersebut tidak dilemparkan lagi ke *method* pemanggil berikutnya. Namun, jika *method* first() tidak mempunyai block catch() yang sesuai, maka *method* berikutnya, yaitu main() akan dicek. Jika method main() juga tidak menangani *exception* ini, maka *exception* tersebut akan dicetak ke dalam output standar dan program berhenti bejalan.

4.2.3 Statement finally

Saat suatu *exception* dilemparkan, alur program dalam suatu *method* membuat jalur yang cenderung tidak linier, melompati baris-baris tertentu, bahkan mungkin akan keluar sebelum waktunya pada kasus di mana tidak ada blok catch yang cocok. Kadang-kadang perlu dipastikan bahwa bagian program yang diberikan akan berjalan, tidak peduli *exception* apa yang terjadi dan yang di-*catch*. Keyword finally dapat digunakan untuk menentukan bagian program seperti itu. Bahkan jika tidak ditemukan bagian catch yang cocok, bagian finally akan dieksekusi sebelum program setelah akhir keseluruhan bagian try. Setiap pernyataan try membutuhkan sekurangnya satu bagian catch atau finallly yang cocok. Setiap kali suatu *method* akan kembali ke pemanggilnya, melalui *exception* yang tidak di-*catch*, atau melalui pernyataan return, bagian finally dieksekusi sebelum *method* kembali. Ini sangat praktis untuk menutup penanganan file dan membebaskan sejumlah *resource* lainnya yang mungkin telah dialokasikan di awal program. Berikut ini merupakan contoh program yang menunjukkan beberapa *method* yang keluar dengan berbagai cara, dan semuanya mengeksekusi bagian finally.

```
public class TryFinally {
  static void methodA() {
    try {
      System.out.println("inside methodA");
      throw new RuntimeException();
    } finally {
       System.out.println("inside finally methodA");
    }
  }
```

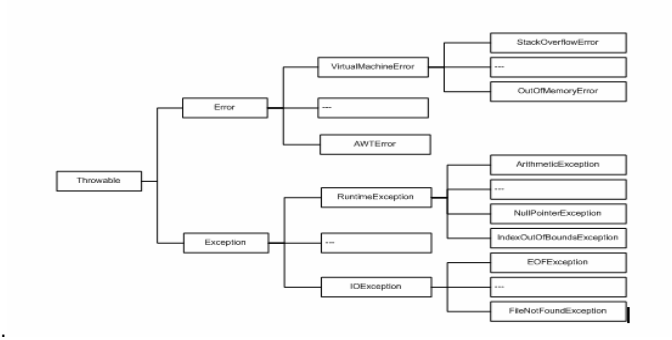
```
  static void methodB() {
    try {
      System.out.print("inside methodB");
      return;
      } finally {
        System.out.print("inside finally methodB");
      }
  }

  public static void main(String[] args) {
    try {
    methodA();
    } catch(Exception e) {
      methodB();
    }
  }
}
```

Pada contoh ini, `methodA` keluar dari blok `try` dengan melemparkan suatu *exception*, lalu bagian `finally` dieksekusi saat keluar. Mirip dengan hal tersebut, statement `try` pada `methodB` keluar melalui pernyataan `return`, lalu bagian `finally` dieksekusi sebelum `methodB` keluar. Berikut ini merupakan output dari program ketika dijalankan.

```
inside methodA
inside finally methodA
inside methodB
inside finally methodB
```

4.3 Kategori *Exception*

Pada bagian sebelumnya, telah dikenalkan dengan tiga kategori besar *exception*. Pada gambar berikut ini ditunjukkan hirarki kelas yang ada di dalam kelas-kelas yang tergabung dalam *exception*.



Gambar 4.1 Hirarki dalam kelas `Throwable`

Kelas `java.lang.Throwable` merupakan kelas induk dari semua obyek yang dapat dilemparkan dan di-*catch* menggunakan mekanisme penanganan *exception*. *Method-method* yang terdapat di dalam kelas `Throwable` mengambil pesan kesalahan yang berkaitan dengan satu *exception* dan mencetak *stack trace* untuk menunjukkan di mana *exception* terjadi. Ada tiga *subclass* penting dari kelas `Throwable`, yaitu kelas `Error`, `RuntimeException`, dan `Exception`.

Sebaiknya kelas `Throwable` tidak digunakan, melainkan menggunakan *subclass* dari *exception* untuk mendeskripsikan satu *exception* tertentu. Berikut ini ditunjukkan penggunaan dari masing-masing kelas dalam *exception* :

- `Error`. Mengindikasikan problem yang fatal yang sulit ditangani. Contoh: *out of memory*. Program tidak perlu menangani kondisi yang demikian.
- `RuntimeException`. *Exception* ini mengindikasikan kondisi yang seharusnya tidak terjadi jika program berjalan dengan normal. *Exception* `ArrayIndexOutOfBoundsException` sebagai contoh tidak akan terjadi jika `Array` dengan index yang tidak melebihi batas `Array` tersebut diakses. Hal yang sama juga berlaku ketika merujuk sebuah variabel obyek yang bernilai `null`. *Exception* yang demikian dapat juga tidak ditangani mengingat program yang didisain dan diimplementasi secara benar tidak pernah menghasilkan *exception* tersebut.
- *Exception* yang lain mengindikasikan permasalahan yang muncul pada saat *runtime* yang biasanya disebabkan oleh hal-hal yang berkaitan dengan lingkungan (environment) di mana program berjalan dan *exception* jenis ini dapat ditangani. Contoh yang termasuk ke dalam kategori *exception* ini adalah *exception* karena file yang tidak dapat ditemukan atau URL yang tidak valid, dua hal yang biasanya dapat terjadi karena *user* salah mengetikkan sesuatu secara keliru.

4.4 *Common Exception*

Beberapa *exception* yang biasa terjadi adalah :

- `ArithmeticException` – terjadi ketika program melakukan operasi pembagian dengan nol pada bilangan integer.

```
int i = 12 / 0;
```

- `NullPointer`*`Exception`* – terjadi ketika program berusaha mengakses atribut atau *method* dari sebuah obyek menggunakan variabel yang tidak merujuk ke sebuah obyek. Contohnya, ketika variabel referensi obyek belum diinisialisasi atau ketika tidak ada obyek yang diinstansiasi.

```
Date d = null;
System.out.println(d.toString());
```

- NegativeArraySize*Exception* – terjadi ketika program membuat Array dengan indeks negatif.
- ArrayIndexOutOfBounds*Exception* – terjadi ketika program mengakses elemen Array melebihi ukuran Array yang bersangkutan.
- Security*Exception* – biasanya dilemparkan di dalam browser, kelas SecurityManager melemparkan *exception* ketika sebuah *applet* berusaha melakukan satu operasi yang membahayakakn *host* atau file-filenya. Berikut ini adalah beberapa contoh hal yang dapat mengakibatkan *exception security* :

  - Mengakses sistem file lokal.
  - Membuka *soket* pada *host* yang berlainan dengan *host* yang melayani *applet.*

4.5 Aturan Deklarasi dan Penanganan *Exception*

Untuk memastikan penulisan kode program yang baik, bahasa Java menetapkan bahwa jika sebuah *checked exception* (yaitu *subclass* dari kelas Exception tetapi bukan merupakan *subclass* dari kelas RuntimeException) terjadi di dalam kode program, maka *method* tempat terjadinya *exception* harus mendefinisikan secara eksplisit tindakan apa yang harus diambil jika *exception* muncul. Berikut hal yang harus diperhatikan :

- Tulis blok try {} dan catch() {} di sekitar kode yang kemungkinan besar akan menghasilkan *exception*. Blok catch() {} harus memuat nama *exception* yang sesuai dengan *exception* yang terjadi atau *superclass* dari *exception* bersangkutan. Dengan melakukan hal tersebut, *exception* sudah ditangani meskipun blok catch() {} dibiarkan kosong.
- Membuat method tempat *exception* terjadi menyatakan bahwa ia tidak akan menangani *exception* yang terjadi. Dengan cara demikian, *exception* akan dilemparkan ke method pemanggil. Hal ini juga berarti *exception* yang terjadi diseklarasikan dan tanggung jawab kepada *method* pemanggil apakah ia akan menangani atau melempar kembali *exception* tersebut didelegasikan.

Sebuah *method* dapat mendeklarasikan sebuah *exception* dengan menggunakan *keyword* throws sebagai berikut:

```
void trouble() throws IOException {}
```

Keyword throws di atas diikuti oleh daftar semua *exception* yang akan dilempar ke *method* pemanggil. Dalam contoh di atas, hanya

dideklarasikan satu *exception*. Pada *method* juga dapat dideklarasikan lebih dari satu *exception* dengan memisahkan antara satu *exception* dengan *exception* yang lain dengan tanda koma sebagaimana ditunjukkan berikut ini :

```
void trouble() throws IOException, OtherException {}
```

Pilihan untuk menangani atau mendeklarasikan *exception* yang terjadi tergantung dari *programmer* dengan mempertimbangkan mana yang lebih layak (method tempat *exception* terjadi atau method pemanggil) untuk menangani *exception* tersebut.

4.6 *Overriding Method* dan *Exception*

Ketika meng-*override* sebuah *method* yang melemparkan *exception*, maka *method* yang meng-*override* tersebut hanya dapat mendeklarasikan *exception* yang berasal dari kelas yang sama atau *subclass* dari *exception* tersebut. Sebagai contoh, jika *method* dalam *superclass* melemparkan IOException, maka method yang meng-*override* dapat melempar *IOException* dan FileNotFoundException (*subclass* dari IOException), tetapi tidak dapat melempar *Exception* (*superclass* dari IOException).

Pada contoh berikut ini dideklarasikan tiga kelas, yaitu TestA, TestB1, dan TestB2. TestA merupakan *superclass* dari kelas TestB1 dan TestB2.

```
public class TestA {
  public void methodA() throws IOException {
    //do some number crunching
  }
}

public class TestB1 extends TestA {
  public void methodA() throws EOFException {
    //do some number crunching
  }
}

public class TestB2 extends TestA {
  public void methodA() throws Exception {
    //do some number crunching
  }
}
```

Kelas TestB1 akan berhasil dikompilasi karena EOFException merupakan *subclass* dari IOException. Tetapi kelas TestB2 tidak berhasil dikompilasi karena *Exception* merupakan *superclass* dari IOException.

5 KELAS-KELAS DASAR DI JAVA

5.1 Kelas IO

Mayoritas dari program yang dibuat membutuhkan interaksi dengan *user*. Interaksi ini dapat dilakukan melalui input teks dan output ke *console* (menggunakan keyboard sebagai input standar dan layar terminal sebagai output standar).

Java 2 SDK mendukung I/O dari *console* dengan menggunakan tiga variabel publik dalam kelas `java.lang.System` :

- `System.out` merupakan sebuah obyek `PrintStream` yang mengacu ke sebuah layar terminal.
- `System.in` merupakan sebuah obyek `InputStream` yang mengacu ke keyboard *user*.
- `System.err` merupakan sebuah obyek `PrintStream` yang mengacu ke sebuah layar terminal.

5.1.1 Menulis ke dalam output strandar

Programmer dapat menulis ke dalam output standar dengan menggunakan *method* `System.out.println(String)`. *Method* yang berasal dari kelas `PrintStream` ini akan mencetak argumen dalam bentuk `String` ke *console* dan menambahkan karakter baris baru di ujung `String`. Method-method berikut ini digunakan untuk mencetak variabel yang berasal dari tipe yang berbeda :

```
void println(boolean)
void println(char)
void println(double)
void println(float)
void println(int)
void println(long)
void println(char[])
void println(Object)
```

Selain itu, dapat digunakan satu set *overloaded method*, yaitu method `print`, yang tidak menambahkan karakter baris baru di ujungnya.

5.1.2 Membaca dari input standar

Kode program berikut ini menunjukkan cara yang dapat digunakan untuk membaca `String` dari *console* input standar :

```
import java.io.*;

public class KeyboardInput {
   public static void main(String[] args) {
    String s;
    //create a buffered reader to read
    //each line from keyboard
    InputStreamReader ir = new InputStreamReader(System.in);
    BufferedReader in = new BufferedReader(ir);
```

Pada baris 5, dideklarasikan variabel dengan tipe `String` yang digunakan untuk menampung setiap baris yang dibaca dari input standar. Baris 8-9 membungkus obyek `System.in` dengan dua obyek pendukung. Obyek `InputStreamReader` (`ir`) membaca karakter dari bentuk *raw bytes* dan mengubahnya ke dalam bentuk karakter *Unicode*. Obyek `BufferedReader` (`in`) menyediakan method `getLine()` yang memungkinkan program untuk membaca dari input standar baris per baris.

```
System.out.println("unix: type ctrl-d or ctrl-c to exit."+
"\nwindows: type ctrl-z to exit");

 try {
   //read each input line and echo it to the screen
   s = in.readLine();
   while(s != null) {
   System.out.println("read: " + s);
   s = in.readLine();
   }
```

Kode di atas membaca baris teks pertama dari input standar. Loop `while` mencetak baris yang dibaca dan kemudian membaca baris berikutnya. Kode di atas dapat ditulis kembali dengan cara yang berbeda sebagai berikut:

```
while((s = in.readLine()) != null) {
   System.out.println("read: " + s);
}
```

Karena method `readLine` dapat melemparkan *exception* I/O, maka pernyataan di atas harus diletakkan di dalam blok `try-catch`.

```
//close the buffered reader
in.close();
```

Kode di atas menutup *input stream* yang terluar untuk melepaskan sumber daya sistem yang terkait dengan pembuat obyek *stream* ini. Lalu program menangani *exception* I/O yang mungkin muncul seperti ditunjukkan pada penggalan kode berikut.

```
        } catch(IOException e) {//catch any IO exception
            e.printStackTrace();
          }
      }
    }
```

5.1.3 Kelas `File`

5.1.3.1 Membuat sebuah obyek File baru

Kelas `File` menyediakan beragam fasilitas yang berhubungan dengan *file* dan informasinya. Dalam teknologi Java, sebuah direktori adalah salah satu contoh bentuk *file*. Sebuah obyek `File` dapat dibuat untuk merepresentasikan sebuah direktori dan menggunakannya untuk mengidentifikasi *file* lainnya. Berikut ini cara membuat obyek `File`.

```
File myFile;
myFile = new File("myfile.txt");
myFile = new File("MyDocs", "myfile.txt");

                atau

File myDir = new File("MyDocs");
myFile = new File(myDir, "myfile.txt");
```

Bentuk konstruktor yang digunakan tergantung obyek `File` lainnya yang akan diakses. Sebagai contoh, jika hanya menggunakan satu *file*, maka gunakan bentuk konstruktor yang pertama. Tetapi, jika menggunakan beberapa file sekaligus dari sebuah direktori, maka akan lebih mudah jika menggunakan bentuk yang kedua.

Kelas `File` mendefinisikan *method-method* yang bersifat *platform-independent* untuk memanipulasi *file* yang dikelola oleh sistem *native file*. Tetapi, *method-method* ini tidak dapat digunakan untuk mengakses isi *file*.

5.1.3.2 *Method* untuk pengujian dan pengecekan kelas `File`

`boolean exists()`
Mengembalikan nilai `boolean` `true` jika *file* atau direktori yang dirujuk oleh obyek `File` tersedia.

boolean isDirectory()
Mengembalikan nilai boolean `true` jika obyek `File` merujuk ke sebuah direktori dan `false` jika sebaliknya.

boolean isFile()
Mengembalikan nilai boolean `true` jika obyek `File` merujuk ke sebuah file dan `false` jika sebaliknya

boolean canRead()
Mengembalikan nilai boolean `true` jika proses pembacaan *file* yang dirujuk oleh obyek `File` diizinkan dan false jika sebaliknya. *Method* ini dapat melemparkan *exception* `SecurityException` jika akses untuk membaca *file* tidak diizinkan.

boolean canWrite()
Mengembalikan nilai boolean `true` jika proses penulisan *file* yang dirujuk oleh obyek `File` diizinkan dan `false` jika sebaliknya. *Method* ini dapat melemparkan *exception* `SecurityException` jika akses untuk menulis ke *file* tidak diizinkan.

5.1.3.3 *Method* untuk mengakses obyek `File`

String getName()
Mengembalikan obyek `String` yang berisikan nama file tanpa *path* nya. Untuk sebuah obyek `File` yang merepresentasikan sebuah direktori, yang dikembalikan adalah nama direktorinya saja.

String getPath()
Mengembalikan obyek `String` yang berisikan *path* ke *file* tersebut termasuk nama *file* atau direktorinya.

String getAbsolutePath()
Mengembalikan obyek `String` yang berisikan *absolute path* dari *file* atau direktori yang dirujuk oleh obyek `File`.

String getParent()
Mengembalikan obyek `String` yang berisikan nama direktori induk (*parent directory*) dari *file* atau direktori yang di-*refer* oleh obyek `File`.

String length()
Mengembalikan nilai (dalam bentuk long) berupa panjang *byte* dari obyek `File` yang bersangkutan.

5.1.3.4 *Method* untuk memodfikasi obyek `File`

`boolean renameTo(File newName)`
method ini digunakan untuk mengubah nama obyek `File` menjadi `newName`.

`boolean mkdir()`
membuat sebuah direktori yang direpresentasikan oleh obyek `File`.

`boolean delete()`
menghapus *file* atau direktori yang dirujuk oleh obyek `File`. Jika proses penghapusan berhasil, *method* ini akan mengembalikan nila `true`.

5.1.4 File Stream I/O

Kode program berikut ini membaca *file* teks dan mencetak setiap baris ke output standar:

```
import java.io.*;
public class ReadFile {
   public static void main(String[] args) {
      //Buat file
      File file = new File(args[0]);

      try {
       //buat sebuah buffered reader untuk baca tiap baris file
          BufferedReader in =
                new BufferedReader(new FileReader(file));
       String s;
```

Pada baris 5, dibuat sebuah obyek `File` dengan menggunakan input dari argumen *command line program*. Pada baris 10, dibuat sebuah *buffered reader* yang membungkus sebuah *file reader*. Kode pada baris ini (konstruktor `FileReader`) akan melemparkan exception `FileNotFoundException` jika file tidak berhasil ditemukan.

```
         s = in.readLine();
         while(s != null) {
           System.out.println("read: " + s);
           s = in.readLine();
         }
         in.close();

      } catch(FileNotFoundException e1) {
         //if this file doesn't exist
         System.err.println("file not found: " + file);

      } catch(IOException e2) {
         //catch any other IO exception
         e2.printStackTrace();
      }
   }
}
```

Blok pada *loop* `while` membaca setiap baris teks dalam *buffered reader* dan mencetaknya ke output standar.

Selanjutnya, *buffered reader* ditutup, yang juga akan menutup *file reader*. Kode penanganan exception akan menangkap *exception* `FileNotFoundException` yang dilempar oleh konstruktor `FileReader`. Beberapa *exception* I/O lainnya ditulis untuk menangani eksepsi yang mungkin saja dilempar oleh *method* `readLine` dan `close`.

Listing program berikut akan membaca input dari keyboard dan memasukkannya ke dalam *file*.

```
import java.io.*;

public class WriteFile{
  public static void main(String[] args) {
    //Buat file
    File file = new File(args[0]);

    try {
      BufferedReader in =
         new BufferedReader(new InputStreamReader(System.in));
      PrintWriter out = new PrintWriter(new FileWriter(file));
      String s;
```

Seperti halnya pada contoh sebelumnya, dibuat sebuah obyek `File` berdasarkan argumen *command line*. Selanjutnya, dibuat sebuah *buffered reader* untuk input standar. Program juga membuat sebuah *print writer* yang membungkus sebuah `FileWriter` untuk menuliskan teks ke suatu file.

```
      System.out.print("Masukkan teks di sini");
      System.out.println("type ctrl-d (or ctrl-z) to stop.");

      while((s = in.readLine()) != null) {
        out.println(s);
      }

      in.close();
      out.close();

     } catch(IOException e) {
        e.printStackTrace();
     }
  }
}
```

Penggalan kode di atas memberikan pesan kepada pengguna untuk memasukkan teks yang akan ditempatkan di dalam file dan mengetikkan Ctrl-D untuk mengakhiri program. Selanjutnya, program menuliskan teks input ke dalam *file*. Akhirnya, *input* dan *output stream* ditutup.

5.2 Kelas Math

Kelas Math di dalam *package* java.lang memuat *method-method* static dan dua konstanta yang digunakan untuk perhitungan matematika. Kelas Math merupakan kelas final dan static sehingga kelas ini tidak dapat diturunkan.

5.2.1 Konstanta

Berikut ini merupakan daftar konstanta yang ada di dalam kelas Math :

- public final static double Math.PI
- public final static double Math.E

5.2.*2 Method-method* dalam kelas Math

Method di dalam kelas Math merupakan *method* statik sehingga untuk mengaksesnya digunakan nama kelasnya. Bentuk umum dari pemanggilan *method* pada kelas Math adalah sebagai berikut :

```
result = Math.aStaticMathMethod();
```

Berikut ini beberapa *method* yang biasa digunakan :

abs()
Method ini mengembalikan nilai absolut dari argumennya. Sebagai contoh :

```
x = Math.abs(99);   //output is 99
x = Math.abs(-99); //output is 99
```

Method ini memiliki versi *overloaded*-nya dengan argumen berupa int, long, float, dan double. Bentuk dari *method* abs adalah sebagai berikut :

```
public static int abs(int a)
public static long abs(long a)
public static float abs(float a)
public static double abs(double a)
```

ceil()
method ini mengembalikan nilai *integer* yang terkecil (dalam bentuk double) yang lebih besar atau sama dengan nilai argumennya. Sebagai contoh, semua pemanggilan *method* di bawah ini menghasilkan nilai double 9.0 :

```
Math.ceil(9.0);   //result is 9.0
Math.ceil(8.8);   //rises to 9.0
Math.ceil(8.02);   //still rises to 9.0
```

Perlakuan terhadap nilai negatif memiliki bentuk yang sama. Yang patut diingat adalah nilai -9 lebih besar dari -10. Sebagai contoh, semua pemanggilan *method* di bawah ini menghasilkan nilai double -9.0 :

```
Math.ceil(-9.0);   //result is -9.0
Math.ceil(-9.4);   //rises to -9.0
Math.ceil(-9.8);   //still rises to -9.0
```

Bentuk dari *method* ceil adalah sebagai berikut :

```
public static double ceil(double a)
```

floor()

Method ini mengembalikan nilai *integer* yang terbesar (dalam bentuk double) yang lebih kecil atau sama dengan nilai argumennya. *Method* ini merupakan lawan dari method ceil. Sebagai contoh, semua pemanggilan method di bawah ini menghasilkan nilai double 9.0 :

```
Math.floor(9.0);   //result is 9.0
Math.floor (9.4);   //drops to 9.0
Math.floor (9.8);   //still drops to 9.0
```

Perlakuan terhadap nilai negatif memiliki bentuk yang sama. Yang patut diingat adalah nilai -9 lebih kecil dari -8. Sebagai contoh, semua pemanggilan *method* di bawah ini menghasilkan nilai double -9.0 :

```
Math.floor(-9.0);   //result is -9.0
Math.floor(-8.8);   //drops to -9.0
Math.floor(-8.1);   //still drops to -9.0
```

Bentuk dari *method* floor adalah sebagai berikut :

```
public static double floor(double a)
```

max()

Method ini menggunakan dua buah argumen dan mengembalikan nilai yang paling besar di antara kedua nilai tesebut. Sebagai contoh :

```
x = Math.max(1024, -5000);   //output is 1024
```

Method ini memiliki versi overloaded-nya dengan argumen berupa int, long, float, dan double. Jika kedua argumen memiliki nilai yang sama, maka method max() akan mengembalikan nilai yang sama

dengan argumen tersebut. Bentuk dari *method* max adalah sebagai berikut :

```
public static int max(int a,int b)
public static long max(long a,long b)
public static float max(float a,float b)
public static double max(double a,double b)
```

min()

Method ini merupakan lawan dari *method* max. *Method* ini menggunakan dua buah argumen dan mengembalikan nilai yang paling kecil di antara kedua nilai tesebut. Sebagai contoh :

```
x = Math.min(0.5, 0.0);   //output is 0.0
```

Method ini memiliki versi *overloaded*-nya dengan argumen berupa int, long, float, dan double. Jika kedua argumen memiliki nilai yang sama, maka *method* min() akan mengembalikan nilai yang sama dengan argumen tersebut. Bentuk dari *method* min() adalah sebagai berikut :

```
public static int min(int a,int b)
public static long min(long a,long b)
public static float min(float a,float b)
public static double min(double a,double b)
```

random()

Method ini mengembalikan nilai *random* (berupa double) yang sama atau lebih besar dari 0.0 dan lebih kecil dari 1.0. *Method* ini tidak menggunakan argumen apapun. Sebagai contoh :

```
x = Math.random();
```

Signature dari *method* random() adalah sebagai berikut :

```
public static double random()
```

round()

Method ini mengembalikan nilai *integer* yang paling dekat dengan nilai argumennya. Algoritma dari *method* ini adalah dengan menambahkan angka 0.5 ke dalam argumen dan membulatkan nilainya ke nilai *integer* terdekat. Jika nilai angka di belakang koma lebih kecil dari 0.5, Math.round() memiliki perilaku sama dengan method Math.floor(). Jika nilai angka setelah angka desimal dari argumennya lebih besar dari 0.5, Math.round() memiliki perilaku sama dengan *method* Math.ceil(). Sebagai contoh :

```
x = Math.round(5.3);   //output is 5
y = Math.round(5.8);   //output is 6
```

Signature dari *method* round adalah sebagai berikut :

```
public static int round(float a)
public static long round(double a)
```

sin()
Method ini mengembalikan nilai *sinus* dari sebuah sudut. Argumen yang digunakan adalah nilai double yang merepresentasikan nilai sudut dalam *radian*. Nilai sudut dalam bentuk derajat (*degree*) dapat diubah nilainya ke dalam bentuk radian dengan menggunakan Math.toRadians(). Sebagai contoh :

```
Math.sin(Math.toRadians(90.0));   //returns 1.0
```

Signature dari *method* sin adalah sebagai berikut :

```
public static double sin(double a)
```

cos()
Method ini mengembalikan nilai *cosinus* dari sebuah sudut. Argumen yang digunakan adalah double yang merepresentasikan nilai sudut dalam *radian*. Sebagai contoh :

```
Math.cos(Math.toRadians(0.0));   //returns 1.0
```

Signature dari *method* cos adalah sebagai berikut :

```
public static double cos(double a)
```

tan()
Method ini mengembalikan nilai *tangen* dari sebuah sudut. Argumen yang digunakan adalah double yang merepresentasikan nilai sudut dalam *radian*. Sebagai contoh :

```
Math.tan(Math.toRadians(45.0));   //returns 1.0
```

Signature dari *method* tan adalah sebagai berikut:

```
public static double tan(double a)
```

sqrt()
Method ini mengembalikan akar dari sebuah nilai yang berbentuk double. Sebagai contoh :

```
Math.sqrt(9.0);   //returns 3.0
```

Signature dari *method* `sqrt` adalah sebagai berikut :

```
public static double sqrt(double a)
```

`toDegrees()`
Method ini menggunakan argumen yang merepresentasikan nilai sudut dalam bentuk radian dan mengembalikannya dalam bentuk derajat (degree). Sebagai contoh :

```
Math.toDegrees(Math.PI * 2.0);   //returns 360.0
```

Signature dari *method* `toDegrees` adalah sebagai berikut :

```
public static double toDegrees(double a)
```

`toRadians()`
Method ini menggunakan argumen yang merepresentasikan nilai sudut dalam bentuk derajat dan mengembalikannya dalam bentuk radian. Sebagai contoh:

```
Math.toRadians(360.0);
//returns 6.283185 which is 2 * Math.PI
```

Signature dari *method* `toDegrees` adalah sebagai berikut :

```
public static double toDegrees(double a)
```

5.3 Kelas `String`

Obyek `String` merepresentasikan urutan karakter *Unicode* yang tidak dapat diubah (*immutable*).

5.3.1 Membuat obyek `String` baru

Berikut ini merupakan beberapa cara untuk membuat obyek `String`.

```
String s = “abc”;
```

Pada contoh di atas, dideklarasikan variabel bertipe `String` dan mengisi obyek `String` yang bernilai “abc” ke dalam variabel `s`.

```
String s = new String(“abc”);
```

Pada contoh kali ini, digunakan konstruktor yang mengambil argumen berupa `String` untuk membuat obyek `String`.

Argumen yang berbeda dapat dimasukkan ke dalam konstruktor String. Pada contoh di atas dimasukkan nilai String sebagai argumen. Untuk lebih lengkapnya, dapat dilihat pada dokumentasi Java.

5.3.2 Beberapa *method* penting di dalam kelas String

Berikut ini merupakan *method-method* di dalam kelas String yang biasa digunakan.

public char charAt(int index)
Method ini mengembalikan karakter yang berada pada indeks tertentu dari String. Hal yang patut diingat adalah indeks dalam String dimulai dari nol. Sebagai contoh :

```
String x = “airplane”;
System.out.print (x.charAt(2));//output is ‘r’
```

public String concat(String s)
Method ini mengembalikan nilai String yang memanggil *method* ini ditambah dengan String s di belakangnya. Sebagai contoh :

```
String x = “taxi”;
System.out.println(x.concat(“cab”));
//output is “taxi cab”
```

Operator + dan += memiliki fungsi yang serupa dengan *method* concat() dalam kelas String. Sebagai contoh :

```
String x = “library”;
System.out.println(x + “ card”);
//output is “library car”
```

atau

```
String x = “Atlantic”;
x += “ ocean”;
System.out.println(x);
//output is “Atlantic ocean”
```

public boolean equalsIgnoreCase(String s)
Method ini mengembalikan nilai boolean (true atau false) yang nilainya tergantung kepada apakah nilai argumen String sama dengan String yang memanggil *method* ini. *Method* ini juga akan mengembalikan nilai true jika kedua String yang dibandingkan huruf besar dan huruf kecilnya berbeda. Sebagai contoh :

```
String x = “Exit”;
System.out.println(x.equalsIgnoreCase(EXIT));// true
System.out.println(x.equalsIgnoreCase(tixe));// false
```

public int length()
Method ini mengembalikan nilai panjang dari String. Sebagai contoh :

```
String x = “01234567”;
System.out.println(x.length());  //return 8
```

public String replace(char old, char new)
Method ini mengubah karakter dari suatu String. Sebagai contoh :

```
String x = “oxoxox”;
System.out.println(x.replace(‘x’, ‘X’));// “oXoXoX”
```

public String substring(int begin)
public String substring(int begin, int end)
Method ini digunakan untuk mengembalikan bagian (*substring*) dari String. Argumen pertama menunjukkan awal dari *substring* (dengan indeks mulai dari nol). Jika pemanggilan *method* ini hanya menggunakan satu argumen, maka string yang dikembalikan adalah mulai dari indeks yang ditunjukkan oleh argumen begin hingga akhir dari String. Sedangkan jika pemanggilan *method* ini memiliki dua argumen, maka *substring* yang dikembalikan adalah mulai dari indeks yang ditunjukkan oleh argumen begin hingga indeks yang ditunjukkan oleh argumen end. Hal yang patut diingat adalah argumen kedua tidak dimulai dari nol, melainkan dimulai dari satu. Sebagai contoh :

```
String x = “0123456789”;
System.out.println(x.substring(5));// “56789”
System.out.println(x.substring(5, 8));// “567”
```

Pada contoh pertama, dimulai dari indeks ke-5 (*zero-based*) hingga akhir dari String. Tetapi, pada contoh kedua dimulai dari indeks ke-5 (*zero-based*), yaitu ‘5’ hingga indeks ke-7 (*not zero-based*), yaitu ‘7’.

public String toLowerCase()
Method ini mengembalikan nilai String yang hurufnya diubah menjadi huruf kecil. Sebagai contoh :

```
String x = “A New Moon”;
System.out.println(x.toLowerCase());// “a new moon”
```

public String toUpperCase()
Method ini mengembalikan nilai String yang hurufnya diubah menjadi huruf besar. Sebagai contoh :

```
String x = "A New Moon";
System.out.println(x.toLowerCase());// "A NEW MOON"
```

public String trim()
Method ini mengembalikan nilai String yang spasi di awal dan di akhirnya dihilangkan. Sebagai contoh :

```
String x = "  hi  ";
System.out.println(x + "x"); // "  hi  x"
System.out.println(x.trim() + "x");// "hix"
```

5.4 Kelas StringBuffer

Obyek StringBuffer merupakan urutan karakter Unicode yang dapat diubah (*mutable*). Dianjurkan untuk menggunakan StringBuffer ketika membuat banyak modifikasi terhadap suatu String.

5.4.1 Konstruktor StringBuffer

Berikut ini merupakan bentuk-bentuk konstruktor yang ada di dalam kelas StringBuffer :

public StringBuffer()
Konstruktor ini membuat sebuah obyek StringBuffer yang kosong. Sebagai contoh :

```
StringBuffer sb1 = new StringBuffer();
```

public StringBuffer(int capacity)
Konstruktor ini membuat sebuah obyek StringBuffer kosong dengan kapasitas sebesar int. Sebagai contoh :

```
StringBuffer sb2 = new StringBuffer(50);
```

Kode di atas akan membuat sebuah obyek StringBuffer yang memiliki kapasitas 50 karakter.

public StringBuffer(String string)
Konstruktor ini membuat sebuah obyek StringBuffer yang berisikan String tertentu. Sebagai contoh :

```
StringBuffer sb3 = new StringBuffer("hello world");
```

Kode di atas akan membuat referensi obyek bertipe `StringBuffer` `sb3` dan menginisialisasinya dengan string "`hello world`".

5.4.2 *Method* yang biasa digunakan dalam kelas `StringBuffer`

Berikut ini merupakan *method-method* yang digunakan untuk memodifikasi obyek `StringBuffer`.

`public synchronizedStringBuffer append(String s)`
Method ini akan memodifikasi `StringBuffer` dengan menambahkan `String` `s` di ujung *buffer*. Kelas `StringBuffer` juga memiliki bentuk *overloaded method* untuk tipe parameter `boolean`, `char`, `char[]`, `double`, `float`, `long`, dan `Object`. Berikut ini merupakan contoh dari penggunaan `StringBuffer`.

```
StringBuffer sb = new StringBuffer("set ");
sb.append("point");
System.out.println( sb ); // "set point"
```

`public synchronized StringBuffer insert(int offset,String s)`
Method ini memodifikasi `StringBuffer` dengan menambahkan `String` `s` pada lokasi *offset* tertentu. Kelas `StringBuffer` juga memiliki bentuk *overloaded method* untuk tipe parameter: `boolean`, `char`, `char[]`, `double`, `float`, `long`, dan `Object`. Sebagai contoh :

```
StringBuffer sb = new StringBuffer("01234567");
sb.insert(4, "---");
System.out.println( sb ); // "0123---4567"
```

`public synchronized StringBuffer reverse()`
Method ini akan membalikkan urutan karakter dari obyek `StringBuffer`. Sebagai contoh:

```
StringBuffer sb = new StringBuffer("A man");
System.out.println( sb.reverse() );
```

5.5 Kelas *Wrapper*

Kelas *Wrapper* memiliki dua fungsi utama, yaitu :

- Menyediakan mekanisme untuk membungkus data primitif ke dalam obyek sehingga data primitif ini dapat menggunakan operasi-operasi yang khusus diperuntukkan untuk obyek, seperti penambahan ke dalam koleksi atau pengembalian dari sebuah *method* yang mengembalikan sebuah obyek.

- Menyediakan satu set fungsi *utility* untuk data primitif. Sebagian besar dari fungsi-fungsi ini berhubungan dengan konversi data primitif dari satu bentuk ke bentuk yang lainnya, sebagai contoh, mengubah data primitif ke dalam bentuk `String` dan sebaliknya, mengubah data primitif ke dalam bentuk basis yang berbeda-beda (*binary, octal*, dan *decimal*).

5.5.1 Sekilas tentang kelas *wrapper*

Setiap data primitif dalam Java memiliki kelas *wrapper* yang bersesuaian. Sebagai contoh, kelas *wrapper* untuk `int` adalah `Integer`, untuk `float` adalah `Float`, dan sebagainya. Yang patut diingat adalah nama dari kelas *wrapper* suatu data primitif adalah sama dengan tipe data primitif yang bersesuaian dengan huruf pertamanya dalam bentuk huruf besar, kecuali untuk tipe data `char` dengan kelas *wrapper* `Character`, dan `int` dengan kelas *wrapper* `Integer`. Tabel di bawah ini menunjukkan tipe data primitif dan kelas *wrapper* yang bersesuaian :

Tabel 5.1
Data Primitif dan Kelas *Wrapper* yang Sejenis Beserta Konstruktornya

TIPE DATA PRIMITIF	KELAS WRAPPER	ARGUMEN KONSTRUKTOR
`boolean`	Boolean	`boolean atau String`
`byte`	Byte	`byte atau String`
`char`	Character	`char`
`short`	Short	`short atau String`
`int`	Integer	`int atau String`
`long`	Long	`long atau String`
`float`	Float	`float atau String`
`double`	Double	`double atau String`

5.5.2 Membuat obyek *wrapper*

Obyek *wrapper* bersifat *immutable*, yang berarti bahwa sekali diberikan nilai pada sebuah obyek *wrapper*, nilainya tidak dapat diubah.

5.5.2.1 Konstruktor Wrapper

Semua kelas *wrapper*, (kecuali `Character`), menyediakan dua buah konstruktor: satu konstruktor menggunakan nilai primitif yang bersesuaian, dan konstruktor yang lain menggunakan `String` yang merupakan representasi dari data primitif yang akan dibungkus di dalam kelas *wrapper*.

```
Integer i1 = new Integer(42);
Integer i2 = new Integer(“42”);
```

atau

```
Float f1 = new Float(3.14);
Float f2 = new Float(“3.14”);
```

Untuk kelas *wrapper* Character, ia hanya menyediakan satu buah konstruktor, yang mengambil nilai char sebagai argumen. Sebagai contoh :

```
Character c1 = new Character(‘c’);
```

5.5.2.2 Method valueOf()

Method statik valueOf() yang disediakan di sebagian besar kelas *wrapper* (kecuali kelas *wrapper* Character) memberikan cara lain untuk membuat obyek *wrapper*. *Method* ini menggunakan String yang merepresentasikan data primitif yang bersesuaian sebagai argumen pertama, dan argumen kedua (jika ada) sebagai basis dari nilai pada argumen pertama. Sebagai contoh:

```
Integer i2 = Integer.valueOf(“100”);
Integer i3 = Integer.valueOf(“101011”, 2);
```

atau

```
Float f2 = Float.valueOf(“3.14f”);
```

5.5.3 Menggunakan utilitas konversi dalam kelas *wrapper*

Berikut ini merupakan *method* yang biasa digunakan dalam kelas *wrapper* untuk mengkonversi suatu nilai.

xxxValue()
Ketika ingin mengubah obyek *wrapper* ke dalam data primitif, dapat digunakan salah satu dari *method* xxxValue() ini. Semua *method* ini tidak menggunakan argumen.

Tabel 5.2
Tabel Method dalam Kelas Wrapper

Method s = static n = NFE exception	Boolean	Byte	Character	Double	Float	Integer	Long	Float
byteValue		x		x	x	x	x	x
doubleValue		x		x	x	x	x	x
floatValue		x		x	x	x	x	x
intValue		x		x	x	x	x	x
longValue		x		x	x	x	x	x
shortValue		x		x	x	x	x	x
parseXxx *s,n*		x		x	x	x	x	x
parseXxx *s,n* (with radix)		x				x	x	x
valueOf *s,n*	x	x		x	x	x	x	x
valueOf *s,n* (with radix)		x				x	x	x
toString	x	x	x	x	x	x	x	x
toString *s* (primitive)	x			x	x	x	x	x
toString *s* (primitive, radix)						x	x	

Sebagai contoh :

```
Integer i2 = new Integer(42);
byte b = i2.byteValue();
short s = i2.shortValue();
double d = i2.doubleValue();

            atau

Float f2 = new Float(3.14f);
short s = f2.shortValue();
System.out.println(s);
```

parseXxx () dan valueOf()

Enam buah method `parseXxx()` (satu untuk setiap tipe *wrapper* numerik) berhubungan erat dengan method `valueOf()` yang ada di semua kelas *wrapper* bertipe numerik (termasuk kelas *wrapper* `Boolean`). Baik method `parseXxx()` maupun `valueOf()` menggunakan `String` sebagai argumennya, melemparkan *exception* `NumberFormatException` jika bentuk `String`nya tidak benar, dan dapat mengubah obyek `String` dari basis yang berbeda-beda jika tipe data primitif yang dirujuknya merupakan satu dari empat bentuk *integer* (`byte`, `short`, `int`, dan `long`). Perbedaan dari kedua *method* ini adalah sebagai berikut :

- `parseXxx()` mengembalikan nilai primitif yang sesuai dengan kelas *wrappernya*.
- `valueOf()` mengembalikan sebuah obyek *wrapper* yang tipenya sama dengan kelas *wrapper* yang memanggil method tersebut.

```
double d4 = Double.parseDouble("3.14");
System.out.println("d4 = " + d4);
Double d5 = Double.valueOf("3.14");
```

toString()

Setiap kelas *wrapper* memiliki method `toString()` yang tidak memiliki argumen dan non-statik. *Method* ini mengembalikan nilai berupa obyek `String` yang merepresentasikan nilai primitif yang dibungkus oleh kelas *wrapper*nya. Sebagai contoh :

```
Double d = new Double("3.14");
System.out.println("d = " + d.toString() );
```

Selain itu, semua kelas *wrapper* yang bertipe numerik mempunyai versi *overloaded* dari *method* `toString()` ini. Masing-masing *method* tersebut mengambil argumen berupa tipe numerik yang bersesuaian (`Double.toString()` mengambill nilai `double`, `Long.toString()` mengambil nilai `long`, dan sebagainya). Sebagai contoh :

```
System.out.println("d = " + Double.toString(3.14);
```

Kelas *wrapper* `Integer` dan `Long` memiliki bentuk ketiga dari *method* `toString()`. *Method* ini bertipe statik, argumen pertamanya adalah nilai primitif yang bersesuaian, dan argumen keduanya merupakan basis. *Method* ini mengembalikan nilai berupa `String` yang merepresentasikan nilai primitif dalam bentuk basis yang sesuai dengan argumen kedua nya. Sebagai contoh :

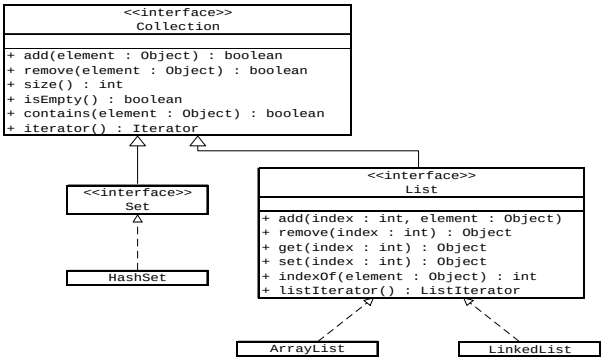
```
System.out.println("hex = " + Long.toString(254,16); //"hex = fe"
```

5.6 Kelas Koleksi

Koleksi merupakan obyek yang merepresentasikan sekumpulan obyek. Obyek yang berada di dalam sebuah koleksi disebut elemen. Koleksi biasanya berkaitan dengan beragam tipe obyek yang kesemuanya merupakan turunan dari tipe obyek tertentu.

- `Collection` – merupakan sekumpulan obyek yang disebut sebagai elemen. Implementasi dari `Collection` ini akan menentukan apakah ada aturan mengenai *ordering* tertentu dan apakah duplikasi elemen diperbolehkan di dalamnya.
- `Set` – merupakan sebuah koleksi yang tidak memiliki aturan *unordered* di mana di dalamnya tidak boleh ada duplikasi elemen.
- `List` – merupakan sebuah koleksi yang memiliki aturan *ordered* di mana di dalamnya duplikasi elemen diperbolehkan.

Semua jenis koleksi memiliki referensi ke obyek-obyek dengan tipe `Object`. Hal ini memungkinkan semua jenis obyek untuk disimpan di dalam koleksi. Implikasi dari elemen dalam koleksi yang bertipe `Object` adalah harus digunakan proses *casting* yang benar ketika mengambil sebuah elemen dari dalam koleksi.



Gambar 5.1 *Interface* `Collection` dan hirarki kelasnya.

Kelas `HashSet` menyediakan implementasi dari *interface* `Set` sementara kelas `ArrayList` dan `LinkedList` menyediakan implementasi bagi *interface* `List`.

5.6.1 Set

Dalam contoh kode berikut ini, dideklarasikan sebuah variabel (`set`) dengan tipe `Set`. Lalu diinisialisasi variabel ini dengan sebuah obyek `HashSet`. Selanjutnya ditambahkan beberapa elemen ke dalamnya dan mencetak isi dari variabel set ke *output* standar.

```
import java.util.*;

public class SetExample {
  public static void main(String[] args) {
    Set set = new HashSet();
    list.add("one");
    list.add("second");
    list.add(new Integer(4));
    list.add("second");   //duplikasi, tidak ditambahkan
    list.add(new Integer(4)); //duplikasi,tidak ditambahkan
    System.out.println(set);
  }
}
```

5.6.2 List

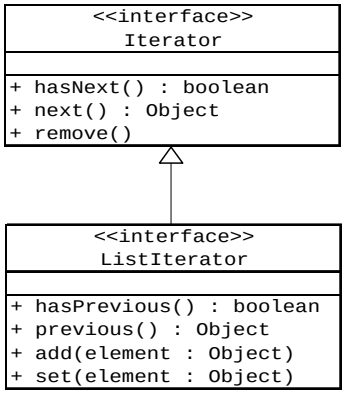
Dalam contoh kode berikut ini, dideklarasikan sebuah variabel (`list`) dengan tipe `List`. Selanjutnya, variabel ini diinisialisasi dengan sebuah obyek `ArrayList`. Selanjutnya ditambahkan beberapa elemen ke dalamnya dan mencetak isi dari variabel list ke output standar. Berbeda dengan contoh sebelumnya, pada contoh ini berhasil dilakukan penambahan elemen yang terduplikasi.

```
import java.util.*;

public class ListExample {
  public static void main(String[] args) {
    List list = new ArrayList();
    list.add("one");
    list.add("second");
    list.add("3rd");
    list.add(new Integer(4));
    list.add(new Float(5.0F));
    list.add("second");          //duplikasi ditambahkan
    list.add(new Integer(4));    //duplikasi ditambahkan
    System.out.println(list);
  }
}
```

5.6.3 Iterator

Sebuah koleksi dapat diiterasi dengan menggunakan iterator. *Interface* `Iterator` memungkinkan untuk meiterasi dengan secara *forward*. Pada kasus iterasi di dalam sebuah obyek `Set`, langkah iterasinya tidak dapat ditentukan sementara langkah iterasi pada sebuah *list* adalah bergerak maju. Sebuah obyek `List` juga mendukung penggunaan `ListIterator` yang memungkinkan proses iterasi secara *backward*. Kode program di bawah ini menggambarkan penggunaan iterator:

```
List list = new ArrayList();
//add some elements
Iterator elements = new Iterator();
while(elements.hasNext()) {
   System.out.println(elements.next());
}
```



Gambar 5.2 Hirarki *interface* Iterator

Method remove digunakan untuk menghapus item di dalam sebuah iterasi. Jika salah satu koleksi tidak mendukung proses penghapusan ini, maka *exception* UnsupportedOperationException akan dilemparkan.

ListIterator bergerak satu arah, yaitu begerak maju dengan menggunakan *method* next() atau mundur dengan menggunakan *method* previous(). Jika menggunakan method previous() setelah menggunakan method next(), maka akan diperoleh elemen sebelum memanggil *method* next(). Begitu juga sebaliknya, apabila digunakan next() setelah previous().

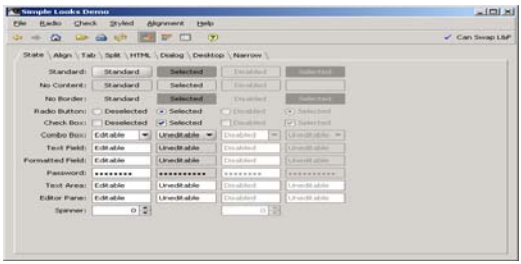
Method set mengubah elemen yang ditunjuk oleh kursor *iterator*. *Method* add menambah elemen baru ke dalam koleksi pada posisi sebelum kursor *iterator*. Dengan demikian, jika memanggil *method* previous seteleh pemanggilan method add, akan diperoleh elemen yang baru dimasukkan ke dalam koleksi tersebut. Jika proses penyetingan dan penambahan elemen tidak didukung oleh koleksi yang bersangkutan, maka UnsupportedOperationException akan dilemparkan.

6

PEMROGRAMAN *GRAPHICAL USER INTERFACE*

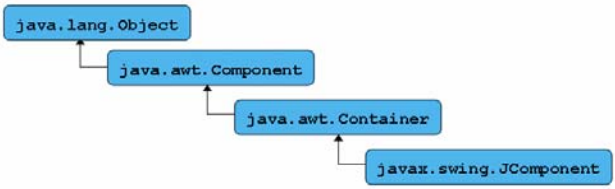
6.1 Pendahuluan

Graphical User Interface (GUI) merupakan *interface* grafis suatu aplikasi yang memfasilitasi interaksi antara pengguna dengan program. Gambar berikut menunjukkan contoh tampilan GUI.



Gambar 6.1 Contoh GUI.

Java memiliki komponen untuk membangun GUI. Salah satu komponen tersebut adalah `Swing`. Komponen `Swing` didefinisikan dalam paket `javax.swing`. `Swing` merupakan komponen GUI yang diturunkan dari *Abstract Windowing Toolkit* dalam paket `java.awt`. Berikut gambar yang menunjukkan hirarki komponen `Swing`.



Gambar 6.2 Hirarki komponen `Swing`.

`JComponent` adalah *superclass* dari semua komponen `Swing`. Sebagian besar fungsionalitas komponen diturunkan dari *superclass* ini. Fungsionalitas tersebut meliputi :

1. *pluggable look* dan *feel*;
2. *shortcut keys* (*mnemonics*), akses langsung ke komponen melalui *keyboard*;
3. *event handling*, penanganan suatu *event*;
4. *tool tips,* teks deskripsi yang muncul ketika *mouse* berada di atasnya.

6.2 JLabel

Label merupakan komponen GUI yang berfungsi untuk menampilkan suatu teks. Teks tersebut biasanya bersifat *read-only*. Umumnya *programmer* jarang sekali mengubah isi suatu label**.** Kelas untuk menampilkan label di GUI berbasis Java bernama `JLabel`. Kelas ini diturunkan dari kelas `JComponent`. Contoh penggunaan `JLabel` ditunjukkan pada gambar berikut.

Gambar 6.3 Contoh `JLabel`.

Suatu teks label dapat diinisialisasi melalui konstruktor `JLabel` sebagaimana ditunjukkan pada penggalan program berikut.

```
label1 = new JLabel("Java Competency Center ITB");
```

atau

```
label 2 = new JLabel("Java Competency Center ITB",
                  ImageIcon, Text_Alignment_CONSTANT);
```

`JLabel` juga dapat menampilkan teks *tool tip*, yaitu teks yang muncul apabila *mouse* berada di atas suatu teks label. Untuk menampilkan *tool tip* gunakan method berikut.

```
myLabel.setToolTipText( "JCC ITB" );
```

Selain inisialisasi teks melalui konstruktor `JLabel`, programmer juga dapat menggunakan method `setText` untuk mengubah atau menyeting teks yang akan ditampilkan dengan menggunakan *method* berikut.

```
myLabel.setText( "Text" );
```

Untuk mengambil isi teks dari JLabel, gunakan *method* berikut.

```
myLabel.getText();
```

Gambar *icon* tertentu juga dapat ditambahkan pada JLabel melalui pemanggilan method berikut ini.

```
Icon bug = new ImageIcon( "pic1.gif" );
label3.setIcon( bug );
```

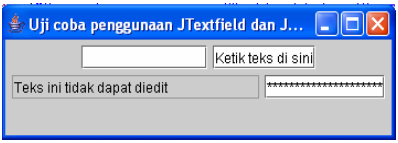
Secara *default*, teks muncul di sebelah kanan *image*. Method setHorizontalTextPosition dan setVerticalTextPosition dapat digunakan untuk menentukan lokasi teks yang akan ditampilkan. Lokasi teks tersebut ditentukan oleh suatu konstanta *integer* yang didefiniskan dalam *interface* SwingConstant pada paket javax.swing. Nilai konstanta tersebut meliputi SwingConstants.LEFT, SwingConstants.RIGHT, SwingConstants.BOTTOM, dan SwingConstants.CENTER. Untuk lebih memahami cara penggunaan JLabel, lihat contoh program berikut.

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

public class LabelTest extends JFrame {
      private JLabel label1, label2, label3;

      public LabelTest(){
             super("Testing Label");
             generateGUI();
      }

      private void generateGUI(){
             Container c = getContentPane();
             c.setLayout(new FlowLayout());
             label1 = new JLabel("Label dengan Text");
             label1.setToolTipText("Ini adalah Label 1");
             c.add(label1);
             Icon bug =
                    new ImageIcon("./image/javalogo.gif");
             label2 = new JLabel("Label dengan text
                       dan icon",SwingConstants.LEFT);
             label2.setToolTipText("Ini adalah label 2");
             c.add(label2);
             label3 = new JLabel();
             label3.setText("Label dengan icon dan
```

```
                        text di bawahnya");
                label3.setIcon(bug);
                label3.setHorizontalTextPosition(
                                        SwingConstants.CENTER);
                label3.setVerticalTextPosition(
                                        SwingConstants.BOTTOM);
                label3.setToolTipText("Ini adalah label 3");
                c.add(label3);
                setSize(275,170);
                show();
        }

        public static void main(String args[]){
                LabelTest app = new LabelTest();

                app.addWindowListener(
                        new WindowAdapter(){
                                public void
                                windowsClosing(WindowEvent e){
                                        System.exit(0);
                                 }
                        }
                );
        }
    }
```

6.3 JTextField dan JPasswordField

JTextFields dan JPasswordField adalah area yang digunakan untuk menampilkan, mengedit, atau menuliskan suatu teks. JTextField diturunkan dari JTextComponent sedangkan JPasswordField diturunkan dari JTextField. Berbeda dengan JTextField, JPasswordField menampilkan teks dalam format asterisk *****. Berikut ini contoh gambar JTextField dan JPasswordField.



Gambar 6.4 Contoh JTextField dan JPasswordField.

JTextField dan JPasswordField memiliki konstruktor berikut :

1. JTextField(10) yang mendefinisikan ukuran teks yang dapat ditulis sebesar 10 kolom.

2. JTextField("Hi") yang mendefinisikan teks "Hi" sebagai tampilan *default* dari JTextField.
3. JTextField("Hi",20) yang mendefinisikan teks dengan tulisan "Hi" sekaligus mendefinisikan ukuran panjang teks yang dapat diisikan sebesar 20 kolom.

JTextField dan JPasswordField memiliki *method* berikut :

1. setEditable(boolean). Jika nilai *boolean* false, maka pengguna tidak dapat mengedit teks pada JTextField, namun masih dapat mengaktifkan suatu *event*.
2. getPassword(). Method ini merupakan method dari kelas JPasswordField. Method ini mengembalikan teks *password* berupa *array* suatu karakter.

JTextField dan JPasswordField memiliki kelas ActionEvent. Method getActionCommand mengembalikan teks yang tertulis pada JTextField. Method getSource mengembalikan sebuah referensi Component. Untuk memahami lebih lanjut tentang penggunaan JTextField dan JPasswordField, pelajari contoh program berikut ini.

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

public class TextFieldTest extends JFrame {
     private JTextField text1, text2, text3;
     private JPasswordField password;
     public TextFieldTest(){
        super("Penggunaan JTextfield dan JPasswordField");
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        text1 = new JTextField(10);
        c.add(text1);
        text2 = new JTextField("Ketik teks di sini");
        c.add(text2);
        text3 = new JTextField("Teks tak dpt diedit",20);
        text3.setEditable(false);
        c.add(text3);
        password = new JPasswordField("Teks tersembunyi");
        c.add(password);
        TextFieldHandler handler = new TextFieldHandler();
        text1.addActionListener(handler);
        text2.addActionListener(handler);
        text3.addActionListener(handler);
        password.addActionListener(handler);
        setSize(325,100);
        show();
     }
```

```
    public static void main(String args[]){
       TextFieldTest app = new TextFieldTest();

       app.addWindowListener(
            new WindowAdapter(){
               public void windowsClosing(WindowEvent e){
                    System.exit(0);
               }
            }
       );
    }

    private class TextFieldHandler
                      implements ActionListener{

       public void actionPerformed(ActionEvent e){
            String s ="";
            if (e.getSource() ==text1)
                    s = "text1 : " + e.getActionCommand();
            else if(e.getSource() == text2)
                    s = "text2 : " + e.getActionCommand();
            else if(e.getSource() == text3)
                    s = "text3 : " + e.getActionCommand();
            else if(e.getSource() ==  password){
                   JPasswordField pwd =
                           (JPasswordField)e.getSource();
                  s = "password : " +
                           new String(pwd.getPassword());
            }
            JOptionPane.showMessageDialog(null,s);
       }
    }
}
```

6.4 JButton

Button merupakan komponen mirip tombol. *Button* terdiri dari beberapa tipe, yaitu command buttons, toggle buttons, check boxes, dan radio buttons.

Command button mengaktifkan ActionEvent ketika diklik. Command button diturunkan dari kelas AbstractButton dan dibuat bersama dengan kelas JButton.

Di atas JButton dapat diletakkan suatu label teks, atau lebih dikenal sebagai *Button label*, dan suatu gambar *icon*. Berikut contoh gambar JButton.

Gambar 6.5 *Button* dengan teks dan *icon*.

Teks Label atau gambar icon yang akan ditulis di atas JButton dapat diinisialisasi melalui konstruktor JButton berikut ini.

```
JButton myButton = new JButton( "Label" );
JButton myButton = new JButton( "Label",myIcon);
```

Method yang sering digunakan dalam JButton adalah sebagai berikut.

```
setRolloverIcon(myIcon);
getActionCommand();
```

Untuk lebih memahami penggunaan JButton pelajari contoh kode program berikut ini.

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

public class ButtonTest extends JFrame {
     private JButton plainButton, fancyButton;

     public ButtonTest(){
            super("Uji Coba Button:");
            generateGUI();
     }

     private void generateGUI(){
            Container c = getContentPane();
            c.setLayout(new FlowLayout());
            plainButton = new JButton("Tanpa Icon");
            c.add(plainButton);
            Icon bug1=new ImageIcon("./image/pic1.gif");
            Icon bug2=new ImageIcon("./image/pic2.gif");
            fancyButton=new JButton("Ada Icon", bug1);
            fancyButton.setRolloverIcon(bug2);
            c.add(fancyButton);

            ButtonHandler handler = new ButtonHandler();
            fancyButton.addActionListener(handler);
```

```
            plainButton.addActionListener(handler);
            setSize(300,300);
            show();
        }

        public static void main(String args[]){
             ButtonTest app = new ButtonTest();

             app.addWindowListener(
                 new WindowAdapter(){
                   public void windowsClosing(WindowEvent e){
                        System.exit(0);
                    }
                 }
             );
        }

        private class ButtonHandler
                        implements ActionListener{

                public void actionPerformed(ActionEvent e){
                     JOptionPane.showMessageDialog(null,
                           "Anda menekan : " +
                              e.getActionCommand());
                }
        }
    }
```

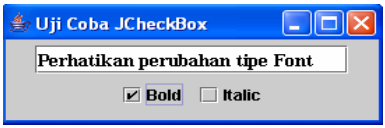
6.5 JCheckBox dan JRadioButton

JToggleButton menurunkan komponen JCheckBox dan JRadioButton. Komponen ini memiliki nilai on/off atau true/false.

Pada kelas JCheckBox, teks ditampilkan di sebelah kanan *checkbox*. Berikut ini cara menginstansiasi obyek JCheckBox.

```
JCheckBox myBox = new JCheckBox( "Judul" );
```

Ketika diklik, JCheckBox mengubah nilai ItemEvent. Pengubahan nilai ItemEvent ini ditangani oleh *interface* ItemListener yang mendefinisikan *method* itemStateChanged. *Method* getStateChange pada kelas ItemEvent mengembalikan nilai *integer* ItemEvent.SELECTED atau ItemEvent.DESELECTED. Gambar berikut ini menunjukkan contoh tampilan JCheckBox.



Gambar 6.6 Contoh tampilan JCheckBox.

Untuk lebih memahami penggunaan JcheckBox, pelajari contoh kode program berikut.

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

public class CheckBoxTest extends JFrame {
      private JTextField t;
      private JCheckBox bold, italic;

      public CheckBoxTest(){
         super ("Uji Coba JCheckBox");
         Container c = getContentPane();
         c.setLayout(new FlowLayout());
         t = new JTextField("Perubahan Font", 20);
         t.setFont(new Font("TimesRoman", Font.PLAIN,14));
         c.add(t);
         bold = new JCheckBox("Bold");
         c.add(bold);
         italic = new JCheckBox("Italic");
         c.add(italic);
         CheckBoxHandler handler = new CheckBoxHandler();
         bold.addItemListener(handler);
         italic.addItemListener(handler);

         addWindowListener(
            new WindowAdapter(){
              public void windowsClosing(WindowEvent e){
                     System.exit(0);}
             }
         );

         setSize(275,100);
         show();
      }

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

```
        private class CheckBoxHandler
                        implements ItemListener{
           private int valBold = Font.PLAIN;
           private int valItalic = Font.PLAIN;

           public void itemStateChanged(ItemEvent e){
              if(e.getSource()== bold)
                  if (e.getStateChange()==ItemEvent.SELECTED)
                        valBold = Font.BOLD;
                  else valBold = Font.PLAIN;
              if(e.getSource()== italic)
                  if (e.getStateChange()==ItemEvent.SELECTED)
                        valBold = Font.ITALIC;
                  else valBold = Font.PLAIN;
              System.out.println(valBold + valItalic);
              t.setFont(new Font("TimesRoman",
                                 valBold + valItalic, 14));
              t.repaint();
           }
        }
  }
```

Berbeda dengan JCheckBox, JRadioButtons memiliki dua status yaitu selected dan deselected. Pada umumnya *radio button* ditampilkan dalam sebuah grup. Hanya satu *radio button* di dalam suatu grup yang dapat dipilih pada satu waktu. Pemilihan satu button menyebabkan button lain berstatus off. Gambar berikut menunjukkan contoh tampilan JRadioButton.

Gambar 6.7 Contoh tampilan JradioButton.

Kelas JRadioButton memiliki konstruktor berikut.

```
        JRadioButton myRadioButton =
                        new JRadioButton("Label",selected);
```

Jika nilai parameter selected adalah true, maka status awal JRadioButton adalah selected. Sama halnya dengan JCheckBox, JRadioButton juga dapat mengaktifkan suatu ItemEvents. Berikut ini penggalan program yang menggabungkan sebuah instan JRadioButton pada sebuah grup.

```
        ButtonGroup myGroup = new ButtonGroup();
        myGroup.add(myRadioButton);
```

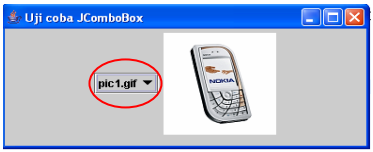
Untuk lebih memahami penggunaan JRadioButton, pelajari contoh program berikut ini.

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

 public class RadioButtonTest extends JFrame {
     private JTextField t;
     private Font plainFont, boldFont;
     private Font italicFont, boldItalicFont;
     private JRadioButton plain, bold, italic, boldItalic;
     private ButtonGroup radioGroup;

     public RadioButtonTest(){
        super("Uji Coba RadioButton");
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        t = new JTextField("Perhatikan
                           perubahan jenis huruf",30);
        c.add(t);
        plain = new JRadioButton("Plain", true);
        c.add(plain);
        bold = new JRadioButton("Bold", false);
        c.add(bold);
        italic = new JRadioButton("Italic",false);
        c.add(italic);
        boldItalic = new JRadioButton("Bold-Italic",
                                      false);
        c.add(boldItalic);
        RadioButtonHandler handler =
                                  new RadioButtonHandler();
        plain.addItemListener(handler);
        bold.addItemListener(handler);
        italic.addItemListener(handler);
        boldItalic.addItemListener(handler);
        radioGroup = new ButtonGroup();
        radioGroup.add(plain);
        radioGroup.add(bold);
        radioGroup.add(italic);
        radioGroup.add(boldItalic);
        plainFont=new Font("TimesRoman",Font.PLAIN,16);
        boldFont=new Font("TimesRoman",Font.BOLD,16);
        italicFont=new Font("TimesRoman",Font.ITALIC,16);
        boldItalicFont =
          new Font("TimesRoman",Font.BOLD+Font.ITALIC,16);
        t.setFont(plainFont);
        setSize(300,500);
        show();
     }
```

```
    public static void main(String args[]){
       RadioButtonTest app = new RadioButtonTest();
       app.addWindowListener(
            new WindowAdapter(){
               public void windowsClosing(WindowEvent e){
                    System.exit(0);
                    }
               }
       );
    }

    private class RadioButtonHandler
                          implements ItemListener{
       public void itemStateChanged(ItemEvent e){
          if (e.getSource()==plain) t.setFont(plainFont);
          else if (e.getSource()==bold)
                    t.setFont(boldFont);
          else if (e.getSource()==italic)
                    t.setFont(italicFont);
          else if (e.getSource()==bold)
                    t.setFont(boldItalicFont);
                   t.repaint();
       }
    }
}
```

6.6 JComboBox

JComboBox merupakan komponen GUI untuk menampilkan daftar suatu item sebagaimana ditunjukkan pada gambar berikut ini.



Gambar 6.8 Contoh tampilan JComboBox.

Kelas JComboBox juga dapat mengaktifkan ItemEvents. Konstruktornya adalah sebagai berikut :

```
JComboBox (arrayOfNames);
```

Setiap item di JComboBox diberi indeks numerik. Elemen pertama diberi indeks 0 dan elemen tersebut dimunculkan sebagai item yang dipilih pada saat instan JComboBox tampil untuk pertama kalinya. *Method* penting JComboBox adalah sebagai berikut :

- `getSelectedIndex` mengembalikan indeks dari item yang sedang dipilih.

```
myComboBox.getSelectedIndex();
```

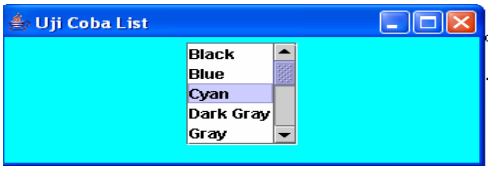
- `setMaximumRowCount(n)` menentukan jumlah maksimum elemen yang ditampilkan ketika pengguna mengklik instan `JComboBox`. `Scrollbar` secara otomatis dihasilkan.

Untuk lebih memahami penggunaan `JComboBox`, pelajari contoh kode program berikut ini.

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

public class ComboBoxTest extends JFrame {
     private JComboBox images;
     private JLabel label;
     private String names[] =
            {"pic1.gif","pic2.gif","pic3.gif","pic4.gif"};
     private Icon icons[] =
            {new ImageIcon("./image/" + names[0]),
             new ImageIcon("./image/" + names[1]),
             new ImageIcon("./image/" + names[2]),
             new ImageIcon("./image/" + names[3])};

     public ComboBoxTest(){
        super ("Uji coba JComboBox");
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        images = new JComboBox(names);
        images.setMaximumRowCount(3);
        images.addItemListener(new ItemListener(){
             public void itemStateChanged(ItemEvent e){
                label.setIcon(
                     icons[images.getSelectedIndex()]);
             }
        });
        c.add(images);
        label = new JLabel(icons[0]);
        c.add(label);
        setSize(400,400);
        show();
     }

     public static void main(String args[]){
            ComboBoxTest app = new ComboBoxTest();
     }
 }
```

6.7 JList

Berbeda dengan JComboBox, JList menampilkan deretan item yang dapat dipilih satu atau lebih. Berikut contoh gambar JList.



Gambar 6.9 Contoh tampilan JList.

Kelas JList memiliki konstruktor berikut ini.

```
JList(arrayOfNames)
```

JList mengambil *array* obyek String. Salah satu *method* JList penting adalah sebagai berikut.

```
setVisibleRowCount(n)
```

menampilkan jumlah n item pada satu waktu, namun tidak secara otomatis membuat *scrolling*. Untuk membuat *scrolling*, gunakan komponen JscrollPane berikut ini.

```
c.add(new JScrollPane(colorList));
```

Method JList lain adalah setSelectionMode(selection_CONSTANT) yang mengambil parameter berikut :

- SINGLE_SELECTION yang berarti hanya satu item yang dapat dipilih.
- SINGLE_INTERVAL_SELECTION yang berarti beberapa item yang berurutan dipilih sekaligus.
- MULTIPLE_INTERVAL_SELECTION yang berarti beberapa item dapat dipilih secara acak.
- getSelectedIndex()mengembalikan nilai indeks dari item yang terpilih.

Event handler di JList mengimplementasikan *interface* ListSelectionListener pada paket javax.swing.event. *Interface* ini mendefinisikan method valueChanged. Untuk mengaktifkan *Event handler* pada Jlist, panggil *method* addListSelectionListener.

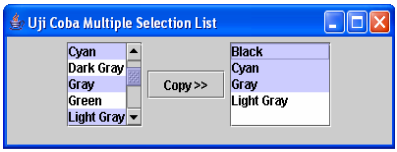
Untuk lebih memahami penggunaan JList, pelajari contoh kode program berikut.

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

public class ListTest extends JFrame {
   private JList colorList;
   private Container c;

   private String colorNames[] = {
       "Black","Blue","Cyan","Dark Gray",
       "Gray","Green","Light Gray","Magneta",
      "Orange","Pink","Red","White","Yellow"};

   private Color colors[] = {
       Color.black,Color.blue,Color.cyan,
       Color.darkGray,Color.gray,Color.green,
       Color.lightGray,Color.magenta,Color.orange,
       Color.pink, Color.red, Color.white,Color.yellow};

   public ListTest(){
       super("Uji Coba List");
       c = getContentPane();
       c.setLayout(new FlowLayout());
       colorList = new JList(colorNames);
       colorList.setVisibleRowCount(5);
       colorList.setSelectionMode(ListSelectionModel.
                                  SINGLE_SELECTION);
       c.add(new JScrollPane(colorList));
       colorList.addListSelectionListener(
         new ListSelectionListener(){
            public  void  valueChanged(ListSelectionEvent  e){
                     c.setBackground(colors[colorList.
                                    getSelectedIndex()]);
          }
         }
       );
       setSize(400,400);
       show();
    }

    public static void main(String args[]){
       ListTest app = new ListTest();
       app.addWindowListener(
          new WindowAdapter(){
              public void windowsClosing(WindowEvent e){
                      System.exit(0);
               }
          });
    }
}
```

Ada dua tipe *multiple selection lists* dalam `JList`, yaitu :

- `SINGLE_INTERVAL_SELECTION`**.** Tipe ini memilih item dengan menekan tombol *Shift*.
- `MULTIPLE_INTERVAL_SELECTION`**.** Tipe ini memilih item dengan menekan tombol *Ctrl key* + klik kanan.

Untuk lebih jelasnya, perhatikan gambar berikut yang menunjukkan tipe *multiple selection list*.



Gambar 6.10 Tampilan `JList` untuk `SINGLE_INTERVAL_SELECTION` dan `MULTIPLE_INTERVAL_SELECTION`.

Beberapa *method* `Jlist` yang penting adalah sebagai berukut :

- `getSelectedValues()`mengembalikan item terpilih.
- `setListData(arrayOfObjects)` menambahkan item pada `JList`.
- `setFixedCellHeight(height)` mendefinisikan tinggi setiap item di `JList` dalam ukuran *pixel*.
- `setFixedCellWidth(width)` mendefinisikan lebar setiap item di `JList` dalam ukuran *pixel*.

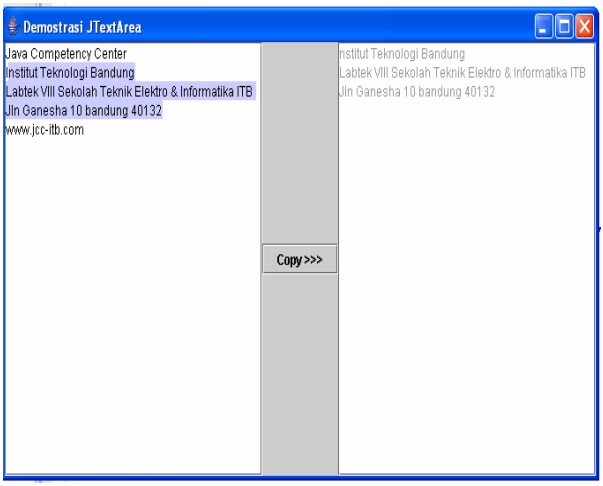
Untuk lebih memahami penggunaan *multiple selection* `Jlist`, pelajari contoh kode program berikut ini.

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

public class MultipleSelectionTest extends JFrame{
   private JList colorList, copyList;
   private JButton copy;

   private String colorNames[] =
          {"Black","Blue","Cyan","Dark Gray",
           "Gray", "Green","Light Gray", "Magneta",
           "Orange", "Pink", "Red", "White", "Yellow"};
```

```
    public MultipleSelectionTest(){
       super("Uji Coba Multiple Selection List");
       Container c = getContentPane();
       c.setLayout(new FlowLayout());
       colorList = new JList(colorNames);
       colorList.setVisibleRowCount(5);
       colorList.setFixedCellHeight(15);
       colorList.setSelectionMode(
        ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
       c.add(new JScrollPane(colorList));
       copy = new JButton("Copy >>");
       copy.addActionListener(
          new ActionListener(){
            public void actionPerformed(ActionEvent e){
               copyList.setListData(
                         colorList.getSelectedValues());
            }
          }
       );
       c.add(copy);
       copyList = new JList();
       copyList.setVisibleRowCount(5);
       copyList.setFixedCellWidth(100);
       copyList.setFixedCellHeight(15);
       copyList.setSelectionMode(
           ListSelectionModel.SINGLE_INTERVAL_SELECTION);
          c.add(new JScrollPane(copyList));
          setSize(400,400);
          show();
       }

    public static void main(String[] args) {
       MultipleSelectionTest apps =
            new MultipleSelectionTest();
       apps.addWindowListener(
            new WindowAdapter(){
                public void windowClosing(WindowEvent e){
                    System.exit(0);
                }
            }
       );
    }
}
```

6.8 JTextArea

JTextArea merupakan komponen yang biasanya digunakan untuk memanipulasi barisan teks. Seperti halnya JTextField, JTextArea diturunkan dari JTextComponent dan memiliki *method* yang sama. Berikut contoh gambar JTextArea.



Gambar 6.11 Contoh tampilan `JTextArea`.

Berikut ini *Method* `JTextField` yang juga dapat digunakan dalam `JTextArea`.

- `getSelectedText` mengembalikan teks terpilih.
- `setText(string)`menuliskan teks pada `JTextArea`.

`JTextArea` memiliki konstruktor sebagai berikut.

```
JTextArea(string, numRows, numColumns)
```

`JTextArea` tidak memiliki *scrolling* otomatis. Oleh karena itu, dibutuhkan obyek `JScrollPane` untuk menambahkan *scrolling bar*. Adapun langkah-langkah untuk menambahkan obyek `JScrollPane` pada `JTextArea` adalah sebagai berikut :

- Instansiasi komponen dengan new `JScrollPane(myComponent)`.
- Tentukan *scrolling policies* `setHorizontalScrollBarPolicy` atau `setVerticalScrollBarPolicy` menggunakan konstanta
  - `JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED`
  - `JScrollPane.VERTICAL_SCROLLBAR_NEVER`
  - `JScrollPane.VERTICAL_SCROLLBAR_ALWAYS`

  Scrolling policy di atas sama untuk *HORIZONTAL*.

Untuk lebih memahami penggunaan JTextArea, pelajari contoh kode program berikut ini.

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

public class TextAreaTest extends JFrame {
      private JTextArea t1, t2;
      private JButton copy;

      public TextAreaTest(){
         super("Demostrasi JTextArea");
         Box b = Box.createHorizontalBox();
         String s = "Java Competency Center\n" +
                    "Institut Teknologi Bandung\n" +
                    "Labtek 7I Sekolah Teknik Elektro &
                     Informatika ITB \n" +
                    "Jln Ganesha 10 bandung 40132\n" +
                    "www.jcc-itb.com";

         t1 = new JTextArea(s,10,15);
         b.add(new JScrollPane(t1));
         copy = new JButton("Copy >>>");
         copy.addActionListener(
           new ActionListener(){
             public void actionPerformed(ActionEvent e){
                    t2.setText(t1.getSelectedText());
             }
           }
         );
         b.add(copy);
         t2 = new JTextArea(10,15);
         t2.setEnabled(false);
         b.add(new JScrollPane(t2));
         Container c = getContentPane();
         c.add(b);
         setSize(400,400);
         show();
      }

      public static void main(String[] args) {
         TextAreaTest app = new TextAreaTest();
         app.addWindowListener(
              new WindowAdapter(){
                 public void windowsClosing(WindowEvent e){
                      System.exit(0);
                 }
              }
         );
      }
   }
```

6.9 Layout Manager

Layout manager menyusun komponen GUI di atas *container*. Penggunaan *layout* memberikan kemudahan dibandingkan dengan menentukan ukuran eksak dan posisi setiap komponen, sehingga para *programmer* akan lebih berkonsentrasi terhadap urusan "*look and feel*" saja.

Pada *layout menager*, komponen diletakkan dari kiri ke kanan lalu ke baris berikutnya. Komponen dapat diletakkan dengan cara rata kiri (*left-aligned*), di tengah (*centered*), atau rata kanan (*right-aligned*). Nilai *dafault*-nya adalah di tengah. Tipe-tipe *layout* dapat berupa `FlowLayout, BorderLayout, GridLayout, atau GridbagLayout.`

6.9.1 `FlowLayout`

`FlowLayout` menyusun komponen dari kanan ke kiri yang selanjutnya ke baris berikutnya. Ukuran *windows* yang diperbesar tidak akan menyebabkan ukuran komponen yang berada di atas `FlowLayout` berubah. Gambar berikut menunjukkan tampilan `FlowLayout`.

Gambar 6.17 Contoh tampilan `FlowLayout`.

Method-method penting dari `FlowLayout` adalah sebagai berikut :

- `setAlignment(position_CONSTANT)` berfungsi untuk menentukan posisi *layout*. Nilai `posititionCONSTANT` dapat berupa `FlowLayout.LEFT`, `FlowLayout.CENTER`, atau `FlowLayout.RIGHT`.
- `layoutContainer(container)` berfungsi untuk me-*update container*.

Untuk lebih memahami penggunaan FlowLayout, pelajari contoh kode program berikut ini.

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

public class FlowLayoutTest extends JFrame {
       private JButton left, center, right;
       private Container c;
       private FlowLayout layout;

       public FlowLayoutTest(){
          super ("Demontrasi penggunaan FlowLayout");
          layout = new FlowLayout();
          c= getContentPane();
          c.setLayout(layout);
          left = new JButton("Left");
          left.addActionListener(
            new ActionListener(){
              public void actionPerformed (ActionEvent e){
                     layout.setAlignment(FlowLayout.LEFT);
                            layout.layoutContainer(c);
              }
            }
           );
           c.add(left);
           center = new JButton("Center");
           center.addActionListener(
             new ActionListener(){
               public void actionPerformed (ActionEvent e){
                  layout.setAlignment(FlowLayout.CENTER);
                  layout.layoutContainer(c);
               }
             }
           );
           c.add(center);
           right = new JButton("Right");
           right.addActionListener(
             new ActionListener(){
               public void actionPerformed (ActionEvent e){
                     layout.setAlignment(FlowLayout.RIGHT);
                     layout.layoutContainer(c);
                }
             });
            c.add(right);
            setSize(500,500);
            show();
       }

       public static void main(String[] args) {
              FlowLayoutTest app = new FlowLayoutTest();
       }
   }
```

6.9.2 BorderLayout

BorderLayout merupakan *default manager* untuk ContentPane. Layout ini menyusun komponen ke dalam 5 wilayah (*region*), yaitu *north*, *south*, *east*, *west*, dan *center*. Komponen-komponen dapat diletakkan pada :

1. *North/South*. Komponen di *region* ini dapat diperluas secara horisontal.
2. *East/West*. Komponen di *region* ini dapat diperluas secara vertikal.
3. *Center*. Komponen di *region* ini dapat diperluas secara vertikal dan horisontal.

Untuk lebih jelasnya, perhatikan contoh gambar tampilan BorderLayout berikut ini.

Gambar 6.13 Contoh tampilan BorderLayout.

Method-method penting yang dapat digunakan pada BorderLayout adalah sebagai berikut.

- Konstruktor BorderLayout(hGap,vGap). Argumen hGap adalah ukuran *gap* horisontal antar *region*. Argumen vGap adalah ukuran *gap* vertikal antar *region*. Nilai *default*-nya adalah 0 baik untuk vertikal maupun horisontal.
- myContainer.add(component,position)menambahkan komponen ke layout. Argumen component menunjukkan komponen yang ditambahkan ke layout, sedangkan argumen position menunjukkan posisi peletakkan komponen, sebagai contoh BorderLayout.NORTH, BorderLayout.SOUTH, BorderLayout.EAST, BorderLayout.WEST, atau BorderLayout.CENTER.

Untuk lebih memahami penggunaan BorderLayout, pelajari contoh kode program berikut ini.

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

public class BorderLayoutTest extends JFrame
                              implements ActionListener{
        private JButton button[];
        private String names[] = {
                "Hie North","Hide South", "Hide East",
                "Hide West", "Hide Center"};
        private BorderLayout layout;

        public BorderLayoutTest(){
                super ("Demonstrasi Border Layout");
                layout = new BorderLayout (5,5);
                Container c = getContentPane();
                c.setLayout(layout);
                button = new JButton[names.length];
                for (int i=0 ; i < names.length; i++){
                        button [i] = new JButton(names[i]);
                        button [i].addActionListener(this);
                }
                c.add(button[0], BorderLayout.NORTH);
                c.add(button[1], BorderLayout.SOUTH);
                c.add(button[2], BorderLayout.EAST);
                c.add(button[3], BorderLayout.WEST);
                c.add(button[4], BorderLayout.CENTER);
                setSize (300,200);
                show();
        }

        public void actionPerformed(ActionEvent e){
                for(int i=0; i<button.length;i++){
                   if(e.getSource()==button[i])
                        button[i].setVisible(false);
                   else button[i].setVisible(true);
                   layout.layoutContainer(getContentPane());
                }
        }

        public static void main(String[] args) {
           BorderLayoutTest app = new BorderLayoutTest();
           app.addWindowListener(
                new WindowAdapter(){
                   public void windowsClosing(WindowEvent e){
                        System.exit(0);}
                  });
        }
}
```

6.9.3 GridLayout

GridLayout membagi Container ke dalam suatu *grid*. Komponen diletakkan dalam suatu baris dan kolom dan memiliki ukuran lebar dan tinggi yang sama. Komponen-komponen ditambahkan mulai dari kiri atas, selanjutnya ke kanan. Jika baris sudah penuh, maka komponen diletakkan di baris selanjutnya, kemudian dari kiri ke kanan. Berikut contoh gambar GridLayout.

Gambar 6.14 Contoh tampilan GridLayout.

Konstruktor GridLayout adalah sebagai berikut.

- GridLayout(rows, columns, hGap, vGap). Konstruktor ini mendefinisikan jumlah baris, kolom, dan ukuran gap horisontal/vertikal antar elemen dalam *pixel*.
- GridLayout(rows, columns). Sama halnya dengan konstruktor pertama, namun dengan nilai default hGap dan vGap sama dengan 0.

Method container validate dapat digunakan untuk merancang ulang (*update*) sebuah *container* yang *layout*-nya sudah diubah. Berikut ini penggalan kode program untuk mengubah *layout* dan meng-*update* Container c jika memenuhi kondisi tertentu.

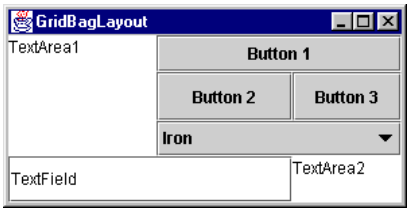
```
public void actionPerformed(ActionEvent e){
       if (toggle) c.setLayout(g1);
       else c.setLayout(g2);
       toggle =! toggle;
       c.validate();
}
```

Untuk lebih memahami penggunaan GridLayout dan penggunaan *method* validate, pelajari contoh kode program berikut ini.

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

public class GridLayoutTest extends JFrame implements
                                    ActionListener {
        private JButton button[];
        private String names[] =
                {"Satu","Dua","Tiga","Empat","Lima","Enam"};
        private boolean toggle = true;
        private Container c;
        private GridLayout g1, g2;

        public GridLayoutTest(){
                super("Demonstrasi GridLayout");
                g1 = new GridLayout(2,3,5,5);
                g2 = new GridLayout(3,2);
                c = getContentPane();
                c.setLayout(g1);
                button = new JButton[names.length];

                for(int i = 0; i<names.length;i++){
                        button[i] = new JButton(names[i]);
                        button[i].addActionListener(this);
                        c.add(button[i]);
                }
                setSize(300,300);
                show();
        }


        public void actionPerformed(ActionEvent e){
                if (toggle) c.setLayout(g1);
                else c.setLayout(g2);
                toggle =! toggle;
                c.validate();
        }

        public static void main(String[] args) {
                GridLayoutTest app = new GridLayoutTest();
                app.addWindowListener(
                  new WindowAdapter(){
                   public void windowsClosing(WindowEvent e){
                        System.exit(0);
                   }
                  }
                );
        }
}
```

6.9.4 GridBagLayout

GridBagLayout merupakan layout manager yang cukup komplek, namun *powerful*. *Layout* ini menyusun komponen ke dalam *grid* sebagaimana ditunjukkan pada gambar berikut ini.



Gambar 6.16 Contoh tampilan GridBagLayout.

Berikut ini langkah-langkah untuk menggunakan GridBagLayout.

1. Gambar terlebih dahulu GUI di kertas.
2. Bagi GUI tersebut ke dalam grid mulai dari baris dan kolom 0. Hal ini dilakukan untuk menempatkan komponen di posisi yang benar.
3. Buat sebuah obyek GridBagConstraints. Objek ini menentukan bagaimana komponen-komponen ditempatkan.
4. Definisikan *instance variable* seperti berikut :
   - gridx - kolom.
   - gridy - baris.
   - gridwidth - jumlah kolom yang dialokasikan.
   - gridheight - jumlah baris yang dialokasikan.
   - weightx - ukuran ruang horisontal.
   - weighty - ukuran ruang vertikal.
5. Inisialisasi nilai weight ke nilai positif (nilai default adalah 0).
6. Definisikan *instance variable fill* dari GridBagConstraints ke NONE (default), VERTICAL, HORIZONTAL, atau BOTH.
7. Definisikan *Instance variable anchor* ke NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, NORTHWEST, atau CENTER (nilai *default*).
8. Masukkan Component dan GridBagConstraints ke *method* setConstraints milik kelas GridBagLayout.

```
gbLayout.setConstraints(c, gbConstraints);
```

9. Setelah Constraints ditentukan, tambahkan Component ke ContentPane.

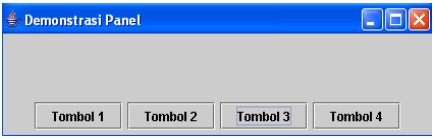
```
container.add(c);
```

Untuk lebih memahami penggunaan GridBagLayout, pelajari contoh kode program berikut ini.

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

public class GridBagTest extends JFrame {
        private Container container;
        private GridBagLayout gbLayout;
        private GridBagConstraints gbConstraints;

        public GridBagTest(){
           super("Demonstrast GridBagLayout");
           container = getContentPane();
           gbLayout = new GridBagLayout();
           container.setLayout(gbLayout);
           gbConstraints = new GridBagConstraints();
           JTextArea ta = new JTextArea("Java", 5,10);
           JTextArea tx = new JTextArea("Competency",2,2);
           String names[] = {"Java", "Competency", "Center"};
           JComboBox cb = new JComboBox(names);
           JTextField tf = new JTextField("Center");
           JButton b1 = new JButton("Java");
           JButton b2 = new JButton("Competency");
           JButton b3 = new JButton("Center");
           gbConstraints.fill = GridBagConstraints.BOTH;
           addComponent(ta,0,0,1,3);
           gbConstraints.fill = GridBagConstraints.NONE;
           addComponent(b1,0,1,2,1);
           addComponent(cb,2,1,2,1);
           gbConstraints.weightx=1000;
           gbConstraints.weighty=1;
           gbConstraints.fill = GridBagConstraints.BOTH;
           addComponent(b2,1,1,1,1);
           gbConstraints.weightx=0;
           gbConstraints.weighty=0;
           addComponent(b3,1,2,1,1);
           addComponent(tf,3,0,2,1);
           addComponent(tx,3,2,1,1);
           setSize(300,200);
           show();
        }

        private void addComponent(Component c, int row,
                                  int column, int w, int h){
                gbConstraints.gridx = column;
                gbConstraints.gridy = row;
                gbConstraints.gridwidth = w;
                gbConstraints.gridheight = h;
                gbLayout.setConstraints(c, gbConstraints);
                container.add(c);
        }

        public static void main(String[] args) {
                GridBagTest app = new GridBagTest();
        }

  }
```

6.10 Panel

Kelas JPanel diturunkan dari JComponent. Setiap JPanel adalah Container. Pada JPanel dapat ditambahkan beberapa komponen atau panel lain. Ukuran area JPanel diukur berdasarkan komponen yang dicakupnya dan terus berkembang untuk mengakomodasi komponen yang ditambahkan padanya. Berikut ini gambar JPanel.



Gambar 6.16 Contoh tampilan JPanel.

Untuk menggunakan JPanel ikuti Langkah-langkah untuk menggunakan JPanel :

1. Buat obyek panel, dan tentukan jenis *layout* dari panel tersebut.
2. Tambahkan komponen ke panel.
3. Tambahkan panel ke ContentPane. Nilai *default*-nya adalah BorderLayout.

Untuk lebih memahami penggunaan JPanel, pelajari contoh kode program berikut ini.

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

 public class PanelTest extends JFrame {
        private JPanel buttonPanel;
        private JButton buttons[];

        public PanelTest(){
                super("Demonstrasi Panel");
                Container c = getContentPane();
                buttonPanel = new JPanel();
                buttons = new JButton[5];
                buttonPanel.setLayout(new FlowLayout());
                for (int i =0; i<buttons.length;i++){
                   buttons[i] = new JButton("Tombol " + (i + 1));
                   buttonPanel.add(buttons[i]);
                }
                c.add(buttonPanel, BorderLayout.SOUTH);
                setSize(500,500);
                show();
        }
```

```
        public static void main(String[] args) {
                PanelTest app = new PanelTest();
                app.addWindowListener(
                   new WindowAdapter(){
                     public void windowsClosing(WindowEvent e){
                         System.exit(0);}
                     }
                );
        }
}
```

JPanel juga dapat digunakan sebagai area untuk menggambar. Sebagaimana ditunjukkan pada gambar berikut ini.

Gambar 6.17 Contoh penggunaan JPanel untuk menggambar.

Ada beberapa hal yang perlu diperhatikan ketika menggunakan JPanel untuk menggambar. Mengkombinasikan Swing GUI dengan menggambar dalam satu *window* dapat menyebabkan *error*. Oleh karena itu, dibutuhkan pemisahan GUI dengan gambar.

Method paintComponent merupakan *method* komponen Swing yang diturunkan dari JComponent. *Method* ini digunakan untuk menggambar. Programmer harus me-*override method* tersebut seperti berikut.

```
public void paintComponent( Graphics g ) {
        super.paintComponent( g );
        // Tambahkan kode untuk menggambar di sini
}
```

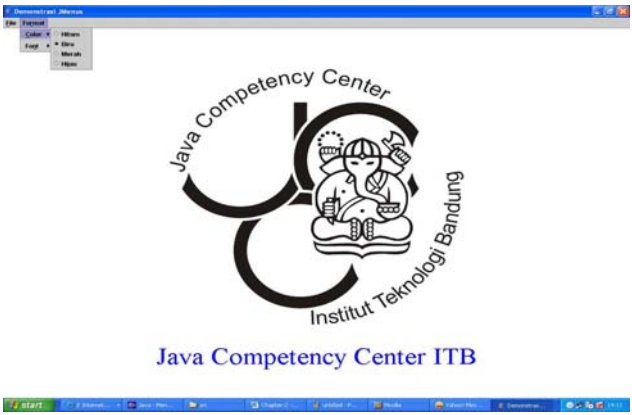
Konstruktor *superclass* dipanggil pada statement pertama, selanjutnya programmer dapat menambahkan kode program untuk menggambar sesuatu sesuai dengan yang diinginkan. Jika konstruktor *superclass* tidak dipanggil, maka *error* akan muncul.

Untuk lebih memahami penggunaan JPanel untuk keperluan menggambar, pelajari contoh kode program berikut ini.

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

public class CustomPanelTest extends JFrame {
      private JPanel buttonPanel;
      private CustomPanel myPanel;
      private JButton circle, square;
      public CustomPanelTest(){
              super("Demonstasi CustomPanel");
              myPanel = new CustomPanel();
              myPanel.setBackground(Color.cyan);
              square = new JButton("Bujur Sangkar");
              square.addActionListener(
                  new ActionListener(){
                       public void actionPerformed(ActionEvent e){
                           myPanel.draw(CustomPanel.SQUARE);
                       }
                  }
              );
              circle = new JButton("Lingkaran");
              circle.addActionListener(
                   new ActionListener(){
                     public void actionPerformed(ActionEvent e){
                       myPanel.draw(CustomPanel.CIRCLE);
                     }
                  }
              );
              buttonPanel = new JPanel();
              buttonPanel.setLayout(new GridLayout(1,2));
              buttonPanel.add(circle);
              buttonPanel.add(square);
              Container c = getContentPane();
              c.add(myPanel, BorderLayout.CENTER);
              c.add(buttonPanel, BorderLayout.SOUTH);
              setSize(300,200);
              show();
      }

      private class CustomPanel extends JPanel {
              public final static int CIRCLE =1, SQUARE = 2;
              private int shape;
              public void paintComponent(Graphics g){
                       super.paintComponent(g);
                       if(shape== CIRCLE) g.fillOval(50,10,60,60);
                       if(shape==SQUARE) g.fillRect(50,10,60,60);
              }
              public void draw(int s){
                       shape=s;
                       repaint();
              }
      }

      public static void main(String[] args) {
              CustomPanelTest app = new CustomPanelTest();
      }
}
```

6.11 Menu

Menu merupakan bagian penting dalam GUI. Obyek menu dilampirkan pada obyek kelas yang memiliki *method* `setJMenuBar`. Menu juga mempunyai `ActionEvents`. Gambar berikut menunjukkan contoh tampilan menu.



Gambar 6.18 Contoh tampilan Menu.

Berikut ini kelas-kelas yang digunakan untuk mendefinisikan menu :

1. `JMenuBar` merupakan `Container` untuk menu.
2. `JMenuItem` mendefinisikan menu item. `JMenuItem` dapat melakukan aksi atau menjadi submenu. Salah satu *method* `JMenuItem` adalah `isSelected`.
3. `JMenu` mendefinisikan menu yang terdiri dari menu item, lalu menambahkannya ke dalam menu *bars*. `JMenu` dapat ditambahkan ke menu lainnya sebagai submenu. Ketika diklik, `JMenu` menunjukkan daftar menu item.
4. `JCheckBoxMenuItem` memperluas `JMenuItem`.

Menu item dapat diakses menggunakan *Mnemonics* (contoh File). *Method* yang digunakan untuk hal tersebut adalah sebagai berikut.

```
JMenu fileMenu = new JMenu("File");
fileMenu.setMnemonic('F');
```

Berikut langkah-langkah untuk menggunakan menu :

1. Instansiasi obyek JmenuBar, selanjutnya kirim obyek tersebut ke *method* `setJMenuBar` dan tentukan *Mnemonics*-nya.
2. Instansiasi obyek `JMenu` dan tentukan *mnemonic*-nya.
3. Instansiasi obyek `JMenuItem` dan tentukan *event handler*-nya.

4. Jika menggunakan JRadioButtonMenuItem, maka buatlah sebuah obyrk ButtonGroup seperti berikut ini.

```
myGroup = new ButtonGroup();
```

lalu tambahkan JRadioButtonMenuItems ke grup tersebut.

5. Tambahkan menu dengan memanggil *method* add(myItem). Tambahkan *separators* jika diperlukan dengan memanggil *method* addSeparator().
6. Jika membuat submenu, tambahkan submenu tersebut ke menu dengan memanggil *method* add(mySubMenu).
7. Tambahkan menu ke menu bar dengan memanggil *method* add(myMenu).

Untuk lebih memahami penggunaan Menu, pelajari kode program berikut ini.

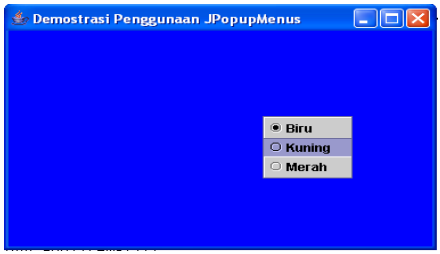
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MenuTest extends JFrame {
      private Color warna[] = {Color.black,Color.blue,Color.red,
                               Color.green};
      private JRadioButtonMenuItem colorItems[], fonts[];
      private JCheckBoxMenuItem styleItems[];
      private JLabel display;
      private ButtonGroup fontGroup, colorGroup;
      private int style;

      public MenuTest(){
              super("Demonstrasi JMenus");
              JMenuBar bar = new JMenuBar();
              setJMenuBar(bar);
              JMenu fileMenu = new JMenu("File");
              fileMenu.setMnemonic('F');
              JMenuItem aboutItem = new JMenuItem("About..");
              aboutItem.setMnemonic('A');
              aboutItem.addActionListener(
                 new ActionListener(){
                   public void actionPerformed(ActionEvent e){

                    JOptionPane.showMessageDialog(MenuTest.this,
                       "Aplikasi ini adalah contoh  penggunaan
                       JMenus","About",
                       JOptionPane.PLAIN_MESSAGE);
                   }
                 });
              fileMenu.add(aboutItem);
              JMenuItem exitItem = new JMenuItem("Exit");
              exitItem.setMnemonic('x');
```

```
exitItem.addActionListener(
  new ActionListener(){
    public void actionPerformed(ActionEvent e){
       System.exit(0);
    }
  });
fileMenu.add(exitItem);
bar.add(fileMenu);
JMenu formatMenu = new JMenu("Format");
formatMenu.setMnemonic('r');
String colors[] = {"Hitam","Biru","Merah","Hijau"};
JMenu colorMenu = new JMenu("Color");
colorMenu.setMnemonic('C');
colorItems=new JRadioButtonMenuItem[colors.length];
colorGroup = new ButtonGroup();
ItemHandler itemHandler = new ItemHandler();

for(int i = 0; i<colors.length;i++){
        colorItems[i] = new
                    JRadioButtonMenuItem(colors[i]);
        colorMenu.add(colorItems[i]);
        colorGroup.add(colorItems[i]);
        colorItems[i].addActionListener(
                                itemHandler);
}
colorItems[0].setSelected(true);
formatMenu.add(colorMenu);
formatMenu.addSeparator();
String fontNames[] =
         {"TimeRoman","Courier","Helvetica"};
JMenu fontMenu = new JMenu("Font");
fontMenu.setMnemonic('n');
fonts = new JRadioButtonMenuItem[fontNames.length];
fontGroup = new ButtonGroup();

for(int i=0; i<fonts.length;i++){
        fonts[i] = new
                JRadioButtonMenuItem(fontNames[i]);
        fontMenu.add(fonts[i]);
        fontGroup.add(fonts[i]);
        fonts[i].addActionListener(itemHandler);
}

fonts[0].setSelected(true);
fontMenu.addSeparator();
String styleNames[] = {"Bold", "Italic"};
styleItems = new
              JCheckBoxMenuItem[styleNames.length];
StyleHandler styleHandler = new StyleHandler();
for(int i=0;i<styleNames.length;i++){
    styleItems[i] = new
                  JCheckBoxMenuItem(styleNames[i]);
    fontMenu.add(styleItems[i]);
    styleItems[i].addItemListener(styleHandler);
}
formatMenu.add(fontMenu);
bar.add(formatMenu);
Icon icon = new ImageIcon("./image/logo.png");
display = new JLabel("Java Competency Center ITB",
                    SwingConstants.CENTER);
```

```
                display.setIcon(icon);
                display.setHorizontalTextPosition(
                                        SwingConstants.CENTER);
                display.setVerticalTextPosition(
                                        SwingConstants.BOTTOM);
                display.setForeground(warna[0]);
                display.setFont(new Font("TimesRoman",
                                      Font.PLAIN,50));
                getContentPane().setBackground(Color.white);
                getContentPane().add(display, BorderLayout.CENTER);

                setSize(500,500);
                show();
        }

        public static void main(String[] args) {
                MenuTest app = new MenuTest();
        }

        private class ItemHandler implements ActionListener{
                public void actionPerformed(ActionEvent e){
                   for(int i = 0;i<colorItems.length;i++)
                      if(colorItems[i].isSelected()) {
                         display.setForeground((warna[i]));
                      }

                   for(int i =0;i<fonts.length;i++)
                      if(e.getSource()==fonts[i]) {
                         display.setFont(new
                                Font(fonts[i].getText(), style, 50));
                      }
                 }
        }

        private class StyleHandler implements ItemListener{
                public void itemStateChanged(ItemEvent e){
                   style = 0;
                   if(styleItems[0].isSelected())
                         style += Font.BOLD;
                   if(styleItems[1].isSelected())
                         style += Font.ITALIC;
                         display.setFont(new
                            Font(display.getFont().getName(),
                                 style, 75));
                         repaint();
                }
        }
}
```

6.12 JPopupMenus

PopupMenus dibuat oleh kelas JPopupMenu. JPopupMenu merupakan turunan dari JComponent. Contoh JPopupMenu ditunjukkan pada gambar berikut.



Gambar 6.19 Contoh tampilan JPopupMenu.

Berikut ini cara mendeklarasikan sebuah obyek JPopupMenu.

```
final JPopupMenu popupMenu = new JPopupMenu();
```

Untuk menampilkan popup ketika *mouse event handler* diaktifkan, biasanya pada saat *method* isPopupTrigger dipanggil, gunakan *method* berikut ini.

```
public void checkForTriggerEvent(MouseEvent e) {
   if(e.isPopupTrigger())
  popupMenu.show(e.getComponent(),e.getX(),e.getY());
}
```

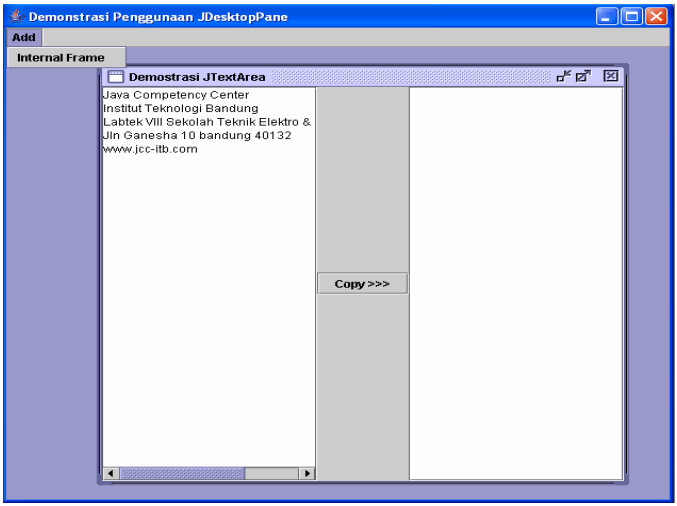
Method show(component,x,y) berfungsi untuk menampilkan JPopupMenu. Argumen component menunjukkan komponen yang diambil dari pemanggilan *method mouse event handler*, getComponent. Argumen x dan y merupakan koordinat JPopupMenu. Untuk lebih memahami penggunaan JpopupMenu, pelajari contoh kode program berikut ini.

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

public class PopupTest extends JFrame {
  private JRadioButtonMenuItem items[];
  private Color colorValues[]={Color.blue,Color.yellow,Color.red};

        public PopupTest(){
            super("Demostrasi Penggunaan JPopupMenus");
            final JPopupMenu popupMenu = new JPopupMenu();
            ItemHandler handler = new ItemHandler();
            String colors[]={"Biru", "Kuning", "Merah"};
            ButtonGroup colorGroup = new ButtonGroup();
            items = new JRadioButtonMenuItem[3];
```

```
        for(int i = 0; i<items.length;i++){
            items[i] = new JRadioButtonMenuItem(colors[i]);
            popupMenu.add(items[i]);
            colorGroup.add(items[i]);
            items[i].addActionListener(handler);
        }

        getContentPane().setBackground(Color.gray);
        addMouseListener(
            new MouseAdapter(){
               public void mousePressed(MouseEvent e){
                    checkForTriggerEvent(e);
                }

                public void mouseReleased(MouseEvent e) {
                     checkForTriggerEvent(e);
                }

                public void checkForTriggerEvent(MouseEvent e) {
                    if(e.isPopupTrigger())
                       popupMenu.show(e.getComponent(),
                                      e.getX(), e.getY());
                }
            }
         );
         setSize(500,500);
         show();
    }

    public static void main(String[] args) {
            PopupTest app = new PopupTest();
            app.addWindowListener(
                new WindowAdapter(){
                 public void windowsClosing(WindowEvent e){
                     System.exit(0);}
                 }
            );
    }

    private class ItemHandler implements ActionListener{
            public void actionPerformed(ActionEvent e){

               for(int i=0;i<items.length;i++){
                  if (e.getSource()==items[i]){
                       getContentPane().setBackground(
                                                colorValues[i]);
                       repaint();
                  }
               }
            }
    }
}
```

6.13 JDesktopPane dan JInternalFrame

Multiple Document Interface (MDI) mendefinisikan *parent window* yang terdiri dari beberapa *child windows*. MDI mengatur beberapa *open document* dan berpindah ke antar dokumen tanpa harus menutupnya. Berikut contoh gambar tampilan MDI.



Gambar 6.20 Contoh tampilan MDI.

Kelas JDesktopPane pada paket javax.swing dan JInternalFrame mendukung MDI.

```
final JDesktopPane theDesktop = new JDesktopPane();
getContentPane().add(theDesktop);
```

JDesktopPane digunakan untuk mengatur JInternalFrame *child windows*. Sama halnya dengan JFrame, JInternalFrame memiliki contentPaint untuk meletakkan komponen. JInternalFrame dapat ditambahkan ke JDesktopPane dengan menggunakan method add(myframe).

```
theDesktop.add(frame);
```

Untuk lebih memahami penggunaan JDesktopPane dan JinternalFrame, pelajari kode program berikut ini.

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

  public class DesktopTest extends JFrame {
        public DesktopTest(){
           super("Demonstrasi Penggunaan JDesktopPane");
           JMenuBar bar = new JMenuBar();
           JMenu addMenu = new JMenu("Add");
           JMenuItem newFrame = new JMenuItem("Internal Frame");
           addMenu.add(newFrame);
           bar.add(addMenu);
           setJMenuBar(bar);
           final JDesktopPane theDesktop = new JDesktopPane();
           getContentPane().add(theDesktop);
           newFrame.addActionListener(
           new ActionListener(){
              public void actionPerformed(ActionEvent e){
                  MyFrame frame = new MyFrame();
                  theDesktop.add(frame);
           }});
           setSize(500,500);
           show();
        }

        public static void main(String[] args) {
                DesktopTest app = new DesktopTest();
        }

        private class MyFrame extends JInternalFrame{
                private JTextArea t1, t2;
                private JButton copy;

                public MyFrame(){
                   super("JTextArea",true,true,true,true);
                   Box b = Box.createHorizontalBox();
                   String s = "Java Competency Center\n" +
                              "Institut Teknologi Bandung;
                   t1 = new JTextArea(s,10,15);
                   b.add(new JScrollPane(t1));
                   copy = new JButton("Copy >>>");
                   copy.addActionListener(
                     new ActionListener(){
                         public void actionPerformed(ActionEvent e){
                            t2.setText(t1.getSelectedText());
                         }
                   });
                    b.add(copy);
                   t2 = new JTextArea(10,15);
                   t2.setEnabled(false);
                   b.add(new JScrollPane(t2));
                   Container c = getContentPane();
                   c.add(b);
                   setSize(500,500);
                   show();
                }
        }
  }
```

6.14 JToolBar

Swing menyediakan komponen baru yang tidak disediakan oleh AWT yaitu JToolBar. Pada dasarnya, JToolBar memiliki kesamaan fungsi dengan JPanel pengikat *button* kecil. Namun, perbedaan utamanya adalah JToolBar bersifat *dockable* atau *floatable*, dimana JToolBar dapat digeser (di-*drag*) ke luar *original window*, sehingga menjadi *standalone window*. JToolBar juga dapat di-*drag* kembali ke dalam window atau dipindahkan posisinya ke samping kiri, kanan, atas atau bawah window. Gambar berikut adalah contoh tampilan JToolbar.

Gambar 6.21 Tampilan JToolbar.

JToolBar dapat dibangun dengan cara memanggil konstruktor kosong (untuk *toolbar* horisontal), atau melalui JToolBar.VERTICAL. Biasanya *toolbar* horisontal diletakkan di area (*region*) utara atau selatan dari *container* yang menggunakan *layout* BorderLayout. Sedangkan *toolbar* vertikal diletakkan di region barat atau timur. Pada *toolbar* dapat diletakkan JButton dan menambahkan *action* padanya. Ada berbagai hal yang perlu diperhatikan, yaitu :

- Tombol pada *toolbar* harus lebih kecil daripada JButton biasa. Oleh karena itu, gunakan *method* setMargin dengan nilai konsturktor obyek Insets nol semuanya sebagaimana ditunjukkan pada penggalan kode program berikut ini.

```
Insets margins = new Insets(0, 0, 0, 0);
for(int i=0; i<toolbarLabels.length; i++) {
    ToolBarButton button =
           new ToolBarButton("./image/" + imageFiles[i]);
    button.setToolTipText(toolbarLabels[i]);
    button.setMargin(margins);
    add(button);
}
```

- Pada JButton label teks disimpan di sebelah kanan *icon*, tetapi teks label pada *toolbar* disimpan di bawah *icon*. Untuk itu, panggil *method* berikut ini.

```
    setVerticalTextPosition(BOTTOM);
    setHorizontalTextPosition(CENTER);
```

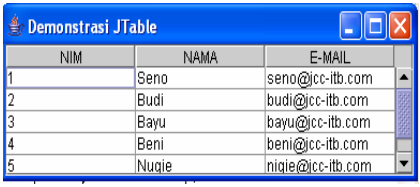
Untuk lebih memahami penggunaan JToolbar, pelajari contoh kode program berikut ini.

```
import java.awt.*;
import javax.swing.*;
import com.jgoodies.looks.plastic.PlasticLookAndFeel;
import java.awt.event.*;

public class JToolBarExample extends JFrame implements ItemListener {
        private BrowserToolBar toolbar;
        private JCheckBox labelBox;

        public JToolBarExample() {
           super("JToolBar Example");
            Container content = getContentPane();
           content.setBackground(Color.white);
           toolbar = new BrowserToolBar();
           content.add(toolbar, BorderLayout.NORTH);
           labelBox = new JCheckBox("Show Text Labels?");
           labelBox.setHorizontalAlignment(SwingConstants.CENTER);
           labelBox.addItemListener(this);
           content.add(new JTextArea(10,30), BorderLayout.CENTER);
           content.add(labelBox, BorderLayout.SOUTH);
           pack();
           setVisible(true);
        }

        public void itemStateChanged(ItemEvent event) {
           toolbar.setTextLabels(labelBox.isSelected());
           pack();
        }

        public static void main(String[] args) {
           try {
              UIManager.setLookAndFeel(new PlasticLookAndFeel());
           } catch (Exception e) {
              e.printStackTrace();
           }
           JToolBarExample app = new JToolBarExample();
              app.addWindowListener(
                 new WindowAdapter(){
                    public void windowsClosing(WindowEvent e){
                          System.exit(0);
                     }
                 });
        }

        private class BrowserToolBar extends JToolBar {
           public BrowserToolBar() {
           String[] imageFiles = {"Left.gif","Right.gif",
                                  "RotCCUp.gif","TrafficRed.gif",
                                  "Home.gif", "Print.gif", "Help.gif"
                                  };
           String[] toolbarLabels = {"Back","Forward","Reload",
                                     "Stop", "Home", "Print", "Help"
                                     };
           Insets margins = new Insets(0, 0, 0, 0);
```

```
            for(int i=0; i<toolbarLabels.length; i++) {
               ToolBarButton button =
                       new ToolBarButton("./image/" + imageFiles[i]);
               button.setToolTipText(toolbarLabels[i]);
               button.setMargin(margins);
               add(button);
            }
         }

         public void setTextLabels(boolean labelsAreEnabled) {
                Component c;
                int i = 0;
                while((c = getComponentAtIndex(i++)) != null) {
                   ToolBarButton button = (ToolBarButton)c;
                   if (labelsAreEnabled)
                      button.setText(button.getToolTipText());
                   else
                      button.setText(null);
                   }
                }
         }
}
```

6.15 JTable

JTabel merupakan komponen untuk menampilkan data dalam tabel. Biasanya komponen ini sering digunakan untuk menampilkan data-data yang diambil dari database. Gambar berikut memperlihatkan contoh penggunaan JTabel.



Gambar 6.23 Contoh JTabel.

Langkah-langkah untuk membuat tabel adalah sebagai berikut :

1. Buat tabel dan model data yang akan ditampilkan di tabel. Hal ini dapat dilakukan dengan cara membuat table dan menghubungkan data model.

```
JTable table = new JTable();
table.setModel (theModel);
```

   atau lebih sederhananya sebagai berikut :

```
JTable table = new JTable(theModel);
```

2. Tampilkan JScrollPane pada tabel dengan cara sebagai berikut :

```
JScrollPane scrollPane = new JScrollPane (table);
```

Jika model data berupa obyek Vector atau Objek Array, masukkan data tersebut ke dalam konstruktor JTable sebagaimana ditunjukkan pada penggalan koe program berikut.

```
String columnNames[] = ...
String data[][] = ...
JTable table = new JTable (data, columnNames);
JScrollPane scrollPane = new JScrollPane (table);
```

Untuk lebih memahami penggunaan JTable, pelajari contoh kode program berikut ini.

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

public class TableDisplay extends JFrame {

        private JTable table;
        private String columnName[] = {"NIM", "NAMA", "E-MAIL"};
        private Object teams[][] =
                {{new Integer(1), "Seno", "seno@jcc-itb.com"},
                {new Integer(2), "Budi", "budi@jcc-itb.com"},
                {new Integer(3), "Bayu", "bayu@jcc-itb.com"},
                {new Integer(4), "Beni", "beni@jcc-itb.com"},
                {new Integer(5), "Nugie", "nigie@jcc-itb.com"},
                {new Integer(6), "Yamin", "yamin@jcc-itb.com"}};

        public TableDisplay(){
                super("Demonstrasi JTable");
                table = new JTable(teams, columnName);
                Container c = getContentPane();
                c.add(new JScrollPane(table), BorderLayout.CENTER);
                setSize(300,300);
                show();
        }

        public static void main(String[] args) {
                TableDisplay app = new TableDisplay();
                app.addWindowListener(
                  new WindowAdapter(){
                     public void windowsClosing(WindowEvent e){
                         System.exit(0);
                     }
                });
        }
}
```

6.16 *Event Handling*

GUI merupakan *event driven*. GUI mengaktifkan sebuah *event* ketika pengguna berinteraksi dengan GUI, sebagai contoh pengguna menggerakan *mouse*, mengklik *mouse*, mengetik suatu teks di sebuah *text field*, dan lainnya. Informasi *event* disimpan di obyek turunan `AWTEvent`.

Berikut langkah-langkah untuk memproses sebuah *event*.

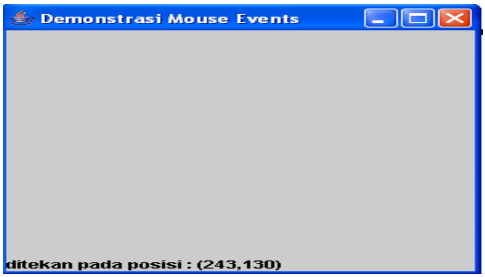
1. Daftarkan sebuah *event listener*. *Event listener* merupakan obyek dari kelas yang mengimplementasikan *event-listener interface* (dari `java.awt.event` atau `javax.swing.event`)
2. Implementasikan *event handler*. *Event handler* adalah *method* yang dipanggil untuk menghasilkan jawaban dari *event* tertentu. *Event handling interface* mempunyai satu atau lebih *method* yang harus didefinisikan.

Proses kerja *event handling* adalah sebagai berikut :

1. Registrasi ke *event listener*. `JComponent` memiliki daftar *Event Listener* yang disebut `listenerList`. Ketika `text1.addActionListener(handler)` dijalankan, entri disimpan di `listenerList`.
2. Menangani *event.* Ketika *event* muncul, *event* tersebut memiliki *event* ID. ID ini digunakan untuk menentukan *method* mana yang harus dipanggil. Jika event berupa `ActionEvent`, maka *method* `actionPerformed` yang akan dipanggil.

6.16.1 *Mouse Event Handling*

Mouse event handling mengambil objek `MouseEvent` yang terdiri dari informasi tentang *event*, termasuk koordinat x dan y dengan pemanggilan `getX` dan `getY`. Gambar berikut memperlihatkan pengambilan obyek `MouseEvent`.



Gambar 6.24 Contoh penggunaan *Mouse event*.

Interface MouseListener dan MouseMotionListener harus mendefinisikan semua *method* berikut :

- *Interface* MouseListener meliputi :
  - public void mousePressed(MouseEvent e)
  - public void mouseClicked(MouseEvent e)
  - public void mouseReleased(MouseEvent e)
  - public void mouseEntered(MouseEvent e)
  - public void mouseExited(MouseEvent e)

- *Interface* MouseMotionListener meliputi :
  - public void mouseDragged(MouseEvent e).
  - public void mouseMoved(MouseEvent e).

Untuk lebih memahami penggunaan *mouse event handling*, pelajari contoh program berikut ini.

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

public class MouseTracker extends JFrame implements MouseListener,
                                                     MouseMotionListener{
        private JLabel statusBar;
        public MouseTracker(){
                super("Demonstrasi Mouse Events");
                statusBar = new JLabel();
                getContentPane().add(statusBar, BorderLayout.SOUTH);
                addMouseListener(this);
                addMouseMotionListener(this);
                setSize(300,300);
                show();
        }

        public void mouseClicked(MouseEvent e){
                statusBar.setText("di-Klik pada posisi : (" +
                                  e.getX() +  "," + e.getY() + ")");
        }

        public void mousePressed(MouseEvent e){
                statusBar.setText("ditekan pada posisi : (" +
                                  e.getX() +  "," + e.getY() + ")");
        }

        public void mouseReleased(MouseEvent e){
                statusBar.setText("dilepas pada posisi : (" +
                                 e.getX() +  "," + e.getY() + ")");
        }

        public void mouseEntered(MouseEvent e){
                statusBar.setText("Mouse di dalam Window");
        }

        public void mouseExited(MouseEvent e){
                statusBar.setText("Mouse di luar Window");
        }
```

```
        public void mouseDragged(MouseEvent e){
                statusBar.setText("di-drag pada posisi : (" +
                                  e.getX() +  "," + e.getY() + ")");
        }

        public void mouseMoved(MouseEvent e){
                statusBar.setText("digerakkan pada posisi : (" +
                                  e.getX() +  "," + e.getY() + ")");
        }

        public static void main(String[] args) {
                MouseTracker app = new MouseTracker();
                app.addWindowListener(
                   new WindowAdapter(){
                      public void windowsClosing(WindowEvent e){
                         System.exit(0);
                       }
                   }
                );
        }
    }
```

6.16.2 Kelas-Kelas Adapter

Mendefinisikan semua *method* suatu *interface* merupakan hal yang kurang efisien jika tidak semua *method* tersebut akan digunakan, sebagai contoh, *programmer* hanya perlu menggunakan satu *method* saja. Solusi untuk masalah ini adalah penggunaan kelas `Adapter`.

Kelas `Adapter` mengimplementasikan sebuah *interface* `Listener`. *Method-method* dalam `Listener` didefinisikan dalam bentuk *empty body*. *Programmer* memperluas (*extend*) kelas `Adapter` dan me-*override method* yang akan digunakannya. Berikut tipe kelas `Adapter` :

ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Perhatikan kode singkat di bawah ini :

```
addMouseMotionListener(
  new MouseMotionAdapter(){
    public void mouseDragged(MouseEvent e){}
  }
);
```

Pada kode di atas, terdapat sebuah *anonymous inner class* yang memperluas (*extend*) MouseMotionAdapter yang sebenarnya

mengimplementasikan MouseMotionListener. *Inner class* mengambil implementasi default (*empty body*) *method* mouseDragged. *Method* tersebut dapat di-*override* sesuai dengan kebutuhan.

Perhatikan kode singkat di bawah ini :

```
Painter apps = new Painter();
apps.addWindowListener(
        new WindowAdapter(){
           public void windowClosing(WindowEvent e){
                System.exit(0);
           }
        }
);
```

Kode singkat di atas digunakan pada sebuah aplikasi yang memperluas (*extend*) JFrame. WindowListener *interface* mendefinisikan tujuh *method*. WindowAdapter mendefinisikan *method-method* tersebut. Pada kasus di atas, di-*override method* windowClosing yang memungkinkan penggunaan tombol close. Untuk lebih memahami penggunaan kelas Adapter, pelajari contoh kode program berikut ini.

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

public class Painter extends JFrame {
        private int xValue = -10, yValue = -10;

        public Painter(){
           super("Contoh program menggambar");
           getContentPane().add(new Label("Drag Mouse"),
                                BorderLayout.SOUTH);

           addMouseMotionListener(
                new MouseMotionAdapter(){
                   public void mouseDragged(MouseEvent e){
                        xValue = e.getX();
                        yValue = e.getY();
                        repaint();
                   }
                }
           );
           setSize(500,500);
           show();
        }

        public void paint(Graphics g){
                g.fillOval(xValue, yValue, 4,4);
        }

        public static void main(String[] args) {
                Painter apps = new Painter();
        }
  }
```

Kelas MouseEvent merupakan turunan dari InputEvent. Kelas ini dapat membedakan antara tombol-tombol pada *multi-button mouse* dan kombinasi *mouse click* dan *keystroke*. Java mengasumsikan bahwa setiap mouse memiliki tombol kiri.

Alt + click = tombol mouse tengah
Meta + click = tombol mouse kanan

Method getClickCount mengembalikan jumlah klik yang dilakukan pengguna pada *mouse*. *Method* isAltDown dan isMetaDown mengembalikan nilai true jika tombol *mouse* tengah atau tombol *mouse* kanan diklik. Untuk lebih jelasnya pelajari kode program berikut.

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

public class MouseDetail extends JFrame{
       private String s="";
       private int xPos, yPos;

       public MouseDetail(){
               super("Mouse Clicks and Button");
               addMouseListener(new MouseClickHandler());
               setSize(300,300);
               show();
       }

       public void paint (Graphics g){
               g.drawString("di-Klik pada posisi : (" + xPos
                        + "," + yPos + ")", xPos, yPos);
       }


       public static void main(String[] args) {
               MouseDetail app = new MouseDetail();
       }

       private class MouseClickHandler extends MouseAdapter {
                public void mouseClicked(MouseEvent e){
                   xPos = e.getX();
                   yPos = e.getY();
                   String s = "Di-klik " +
                                  e.getClickCount() + " kali ";
                   if (e.isMetaDown())
                         s += "dengan tombol mouse kanan";
                   else if (e.isAltDown())
                         s += "dengan tombol mouse tengah";
                   else
                         s += "dengan tombol mouse kiri";
                   setTitle(s);
                   repaint();
                }
       }
}
```

6.16.3 *Keyboard Event Handling*

KeyListener *interface* menangani *key events* (*keys pressed* pada *keyboard*). *Interface* ini mendefinisikan *method* berikut :

1. keyPressed - dipanggil ketika beberapa *key* pada *keyboard* ditekan.
2. keyTyped - dipanggil ketika *non-action key* pada *keyboard* ditekan. Contoh *action key* adalah tanda panah, *home*, *end, page up*, *page down*, *function keys*, *num lock*, *print screen*, *scroll lock*, *caps lock*, *pause*.
3. keyReleased - dipanggil ketika beberapa *key* pada *keyboard* dilepas.

Setiap *method* mengambil KeyEvent sebagai argumennya. KeyEvent ini merupakan *subclass* dari InputEvent. *Method-method* KeyEvent adalah sebagai berikut :

- getKeyCode mengambil kode *key*. Setiap *key* direpresentasikan dengan kode *virtual key*.
- getKeyText mengambil konstanta *kode key* dan mengembalikan nama *key*.
- getKeyChar mengambil karakter *unicode* dari *key* yang ditekan.
- isActionKey mengembalikan nilai true jika *key* berupa *action key*.
- getModifiers mengembalikan *modifiers* yang ditekan.
- getKeyModifierText mengembalikan nama *modifier keys* dengan tipe String.

Untuk lebih jelasnya pelajari kode program berikut. Bila perlu, sempurnakan kode program berikut.

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

public class KeyDemo extends JFrame implements KeyListener {
        private String line1 ="", line2="", line3="";
        private JTextArea textArea;

        public KeyDemo(){
                super("Demonstrasi KEyborad event");
                textArea = new JTextArea(10,20);
                textArea.setText("Tekan tombol di keyboard");
                textArea.setEnabled(false);
                addKeyListener(this);
                getContentPane().add(textArea);
                setSize(350,100);
                show();
        }
```

```
public void keyPressed(KeyEvent e){
   line1 = "Key pressed: " +
            KeyEvent.getKeyText( e.getKeyCode() );
   setLines2and3( e );
}

public void keyReleased(KeyEvent e){
        line1 = "Key Release: " +
                KeyEvent.getKeyText( e.getKeyCode() );
        setLines2and3( e );
}

public void keyTyped(KeyEvent e){
        line1 = "Key Typed: " + e.getKeyChar();
        setLines2and3( e );
}

private void setLines2and3( KeyEvent e ){
   line2 = "This key is " +
               (e.isActionKey()?"":"not ") +
                "an action key";
   String temp =
           KeyEvent.getKeyModifiersText(  e.getModifiers()  );

   line3 = "Modifier keys pressed: " +
           ( temp.equals( "" ) ? "none" : temp );
   textArea.setText(line1+"\n"+line2+"\n" + line3 + "\n" );
}
```

7 MULTITHREADING

7.1 Konsep Dasar

Perkembangan sistem komputer dewasa ini menuntut kinerja komputer yang lebih baik. Salah satu parameter kinerja tersebut misalnya "apakah komputer dapat melakukan banyak proses / tugas dalam waktu yang relatif hampir bersamaan ?" Sebagai contoh, pada sistem komputer saat ini. Sebuah PC dapat melakukan proses "memutar lagu ber-format MP3", "melakukan *download file* dari *server* lain", dan "mengirim data ke *printer* untuk dicetak" dalam waktu bersamaan. Karena dukungan spesifikasi mikroprosesor berkecepatan tinggi, maka proses-proses tersebut dirasakan oleh pengguna dilakukan pada waktu bersamaan, meskipun pada batasan tertentu eksekusi setiap aplikasi akan terasa tidak serentak dengan eksekusi aplikasi lainnya.

7.1.1 Sekilas tentang *Central Processing Unit* (CPU)

Setiap sistem komputer mempunyai unit pemroses pusat (*Central Processing Unit* - CPU). CPU akan menjalankan rangkaian instruksi yang membentuk suatu fungsi tertentu yang mendukung operasional sistem komputer secara keseluruhan. Sebagai contoh, sebuah CPU dapat menjalankan rangkaian instruksi untuk memutar piringan *hard-disk*, menjalankan aplikasi *game*, mengendalikan *peripheral* seperti *printer*, *scanner*, bahkan melakukan komunikasi dengan CPU lain pada sistem komputer yang sama misalnya pada sistem *dual-core*.

Ketika menjalankan setiap instruksi, CPU selalu menyediakan satu alokasi waktu atau slot waktu. Jadi, secara low-level setiap instruksi dijalankan pada waktu yang berbeda.

7.1.2 *Execution context*

Execution context adalah satu entitas yang menggambarkan satu aliran proses. *Execution context* dapat juga disebut sebagai *virtual CPU*. Artinya, pada sebuah *execution context*, sebuah proses yang mirip dengan proses yang dijalankan oleh sebuah CPU secara fisik dijalankan.

Meskipun secara fisik CPU pada sebuah komputer hanya berjumlah 1 atau 2 buah (dual-core), tetapi dengan menerapkan konsep *execution context*, satu CPU dapat berjalan lebih dari 1 *virtual CPU*.

7.2 Membuat *Thread*

Sebuah *thread* menggambarkan sebuah *execution context*. Ketika sebuah *thread* diinstantiasi, *thread* tersebut membutuhkan kode untuk dieksekusi, karena pada dasarnya, *thread* merupakan suatu entitas yang mengakomodasi eksekusi program. Kode yang dibutuhkan oleh *thread* tersebut berada dalam obyek. Oleh karena itu, pada implementasinya sebuah *thread* pasti berkorespondensi dengan sebuah obyek.

Obyek yang datanya akan digunakan dalam *thread* disebut *obyek* Thread. Obyek Thread adalah obyek yang merupakan instanstiasi dari kelas yang merupakan turunan dari kelas Thread, atau mengimplementasi *interface* Runnable.

7.2.1 Membuat obyek Thread menggunakan kelas Thread

Kelas pada teknologi Java yang digunakan untuk menciptakan *thread* adalah kelas Thread. Kelas yang obyek-obyeknya mengandung *method* yang akan dieksekusi dalam sebuah *thread* dapat didefinisikan dengan mendesain kelas tersebut menjadi turunan kelas Thread. Ketika mendefinisikan kelas yang merupakan turunan Thread, kelas tersebut harus meng-override *method* run().

Method run() adalah *method* yang *code block*-nya akan dieksekusi ketika *thread* dijalankan. Pembuatan kelas yang merupakan turunan kelas Thread diperlihatkan pada contoh kode program berikut ini.

```
public class Orang extends Thread{
   private String name;
   private long delay;
   private String alamat;
   private int counter;

   public Orang (String n, String a,long d){
      name = n;
      delay = d;
      alamat = a;
      counter = 0;
   }
```

```
    public  void run(){
       while(true){
          try{
             cetakNama();
             Thread.sleep(delay);
             cetakAlamat();
             counter++;
             if(counter==5){
                break;
             }
          }catch(InterruptedException e){
          }
       }
    }

    public void cetakNama(){
          System.out.println(name + "berkata : Namaku “ +
                             “adalah " + name );
    }

    public void cetakAlamat(){
         System.out.println(name+ "berkata : Alamatku “ +
                             “adalah di " + alamat);
    }
}
```

Pada contoh di atas ditunjukkan kelas Orang yang didefinisikan sebagai turunan dari kelas Thread. Perhatikan *method* run() pada kelas Orang. *Method* ini merupakan *overriding method* milik kelas Thread. Ketika *thread* dijalankan, *method* run() pada obyek Orang akan dijalankan.

7.2.2 *Interface* Runnable

Ada satu masalah ketika membuat kelas yang menjadi turunan dari kelas Thread, yaitu tidak dapat diturunkannya kelas tersebut dari kelas lain. Ingat bahwa satu kelas hanya dapat merupakan turunan dari 1 kelas induk saja padahal kelas yang dibuat dapat saja lebih efisien pendefinisiannya jika didisain sebagai kelas anak dari kelas lain selain Thread. Solusinya adalah dengan mengimplementasikan *interface* Runnable pada kelas yang akan dibuat. *Interface* ini hanya mengandung 1 buah *method*, yaitu run(). Sesuai dengan aturan implementasi *interface*, *method* ini harus didefinisikan ketika kelas didefinisikan. Pembuatan kelas yang mengimplementasi *interface* Runnable diperlihatkan pada contoh kode program berikut ini.

```
public class Orang extends Object implements Runnable{
   private String name;
   private long delay;
   private String alamat;
   private int counter;

   public Orang (String n, String a,long d){
      name = n;
      delay = d;
      alamat = a;
      counter = 0;
   }

   public  void run(){
      while(true){
         try{
            System.out.println(name + "berkata : Namaku " +
                               "adalah " + name );
            Thread.sleep(delay);
            System.out.println(name+ "berkata : Alamatku " +
                                "adalah di " + alamat);
            counter++;
            if(counter==5){
               break;
            }
         }catch(InterruptedException e){
         }
      }
   }
}
```

7.3 Deklarasi, Inisialisasi dan Eksekusi Thread

7.3.1 Deklarasi dan inisialisasi *thread*

Thread merupakan obyek yang menangani eksekusi sebuah proses atau dengan kata lain Thread adalah representasi sebuah *virtual CPU*. Oleh karena itu, sebelum proses yang ditangani dieksekusi terlebih dahulu harus dilakukan deklarasi dan inisialisasi obyek Thread.

Deklarasi dan inisialisasi obyek Thread dapat dilakukan tergantung cara pendefinisian kelasnya, apakah mengimplementasi *interface* Runnable atau menurunkan dari kelas Thread.

Untuk *thread* yang obyeknya diturunkan langsung dari kelas Thread, deklarasi dan inisialisasinya dapat langsung dilakukan dengan memanggil konstruktornya saja. Dengan demikian obyek dan *thread*-nya sudah langsung terbentuk. Deklarasi dan inisialisasi obyek Thread dengan cara ini diperlihatkan pada contoh kode program berikut.

```
public class AplikasiThread1 {
   public static void main(String[] args){
      Orang rudi = new Orang("Rudi", "Medan", 1000L);
   }
}
```

Untuk *thread* yang obyeknya mengimplementasi *interface* Runnable, dapat digunakan konstruktor-konstruktor sebagai berikut :

a. Thread (*Runnable* t)
b. Thread (*Runnable* t, String name).

dengan obyek yang mengimplementasi *interface Runnable* menjadi argumen dari konstruktor tersebut. Deklarasi dan inisialisasi obyek Thread dengan cara tersebut ditunjukkan pada contoh kode program berikut ini.

```
public class AplikasiThread1 {
   public static void main(String[] args){
      Orang rudi = new Orang("Rudi", "Medan" ,
                        1000L);
      Thread t1 = new Thread(rudi);
   }
}
```

7.3.2 Eksekusi *thread*

7.3.2.1 State minimal pada *thread* (Model I)

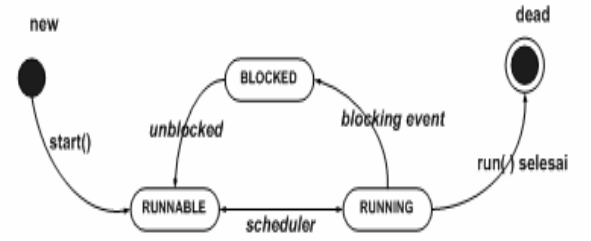
Sebelum masuk pada pembahasan eksekusi *thread*, terlebih dahulu dibahas mengenai state minimal pada *thread*. State ini menggambarkan keadaan *thread* pada saat dieksekusi.

Ketika sebuah *thread* mulai dijalankan, maka *thread* tersebut akan masuk ke dalam state yang disebut **Runnable**. Pada saat *thread* masuk state ini, maka pada saat itu juga *thread* itu masuk ke dalam daftar *scheduler* yang kemudian menunggu untuk dieksekusi. Keputusan sebuah *thread* dieksekusi atau tidak bergantung pada *scheduler*. *Scheduler* akan mengatur penggiliran eksekusi.

Thread yang mendapat kesempatan untuk dieksekusi, akan masuk ke dalam state **Running**. Pada state ini, jika tidak ada interupsi atau delay, *thread* akan dieksekusi sampai pada akhir statement.

Thread yang dieksekusi tanpa interupsi atau *delay* ketika proses yang dibawanya selesai dieksekusi akan masuk ke dalam akhir dari lifecycle, yaitu **Dead**.

Jika pada saat menjalankan eksekusi, suatu *thread* terkena interupsi atau *delay*, maka *thread* tersebut masuk ke dalam state **Blocked**. Pada state ini, *thread* akan menunggu sampai dirinya memenuhi syarat untuk bisa masuk lagi ke state **Runnable**. Misalnya jika *thread* tersebut masuk state **Blocked** karena *delay*, maka kesempatan untuk masuk ke state **Runnable** akan didapatkan ketika delaynya sudah selesai. Untuk lebih jelasnya, perhatikan gambar berikut ini.



Gambar 7.1 *State* minimal pada *thread* (Model I).

7.3.2.2 Implementasi *state* pada eksekusi *thread*

Thread yang telah dibuat tidak serta-merta langsung mengeksekusi proses dalam blok kode dari *method* `run()`. *Thread* ini harus di-*start* terlebih dahulu dengan memanggil method `start()`. Method `start()` ini akan memanggil method `run()` milik obyek. Dengan demikian, *state thread* masuk ke *state* **Runnable**.

Untuk *thread* yang obyeknya merupakan turunan kelas Thread, cara memulainya adalah dengan memanggil method `start()` milik obyek tersebut seperti ditunjukkan pada contoh kode program berikut.

```
public class AplikasiThread1 {
   public static void main(String[] args){
      Orang rudi = new Orang("Rudi", "Medan", 1000L);
      rudi.start();
   }
}
```

Sedangkan untuk *thread* yang obyeknya merupakan implementasi *interface* `Runnable`, cara memulainya adalah dengan memanggil method `start()` milik `Thread`. Contoh kode program berikut menunjukkan cara pemanggilan method `start()` milik `Thread`.

```
public class AplikasiThread1 {
   public static void main(String[] args){
      Orang rudi = new Orang("Rudi", "Medan", 1000L);
      Thread t1 = new Thread(rudi);
      t1.start();
   }
}
```

Ketika *method* `start()` dipanggil, blok kode pada *method* `run()` akan dijalankan. Proses *thread* dapat ditunda selama sekian milidetik dengan pemanggilan *method* berikut.

```
Thread.sleep(delay);
```

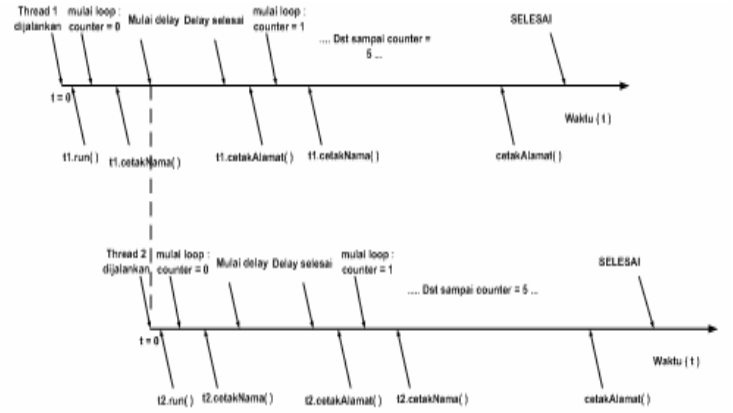
7.3.3 *Multithreading*

Dalam satu aplikasi dapat dibuat lebih dari 1 thread. Fitur ini dinamakan *multithreading*. Pada *multithreading*, *thread-thread* akan dijalankan "seakan-akan" bersamaan. Mengapa "seakan-akan" ? Karena sebenarnya jika diamati pada low level, sebenarnya *thread-thread* tersebut tidak berjalan bersamaan, tetapi karena selisih waktu antara eksekusi *thread* yang satu dengan yang lain terlalu kecil, maka seakan-akan eksekusi *thread-thread* dirasakan bersamaan oleh manusia. Perhatikan contoh kode program berikut ini.

```
public class AplikasiThread1 {
   public static void main(String[] args){
      Orang rudi = new Orang("Rudi", "Medan", 5L);
      Orang stefani = new Orang("Stefani","Bandung",3L);
      Orang ahmad = new Orang("Ahmad", "Jakarta",2L);
      Thread t1 = new Thread(rudi);
      Thread t2 = new Thread (stefani);
      Thread t3 = new Thread(ahmad);
      t1.start();
      t2.start();
      t3.start();
   }
}
```

Pada contoh di atas diperlihatkan 3 buah obyek Orang : *rudi, stefani,* dan *ahmad* yang mempunyai *method* `run()` untuk dieksekusi oleh *thread*. Ketiga obyek tersebut akan dienkapsulasi ke dalam obyek-obyek Thread *t1, t2,* dan *t3*. Setelah obyek-obyek *t1, t2,* dan *t3* selesai diinstanstiasi, maka langkah berikutnya adalah menjalankan *thread-thread* tersebut dengan memanggil *method* `start()` pada masing-masing *thread*.

Ketika pemanggilan *method* `start()`, obyek-obyek *t1, t2,* dan *t3* masuk ke *state* **Runnable** dan menunggu *scheduler* untuk dieksekusi dalam hal ini dimasukkan ke dalam *state* **Running**.



Gambar 7.2 Penggiliran eksekusi di antara 2 *thread*.

Gambar 7.2 memperlihatkan penggiliran eksekusi diantara 2 Thread. Perlu ditegaskan bahwa pada suatu saat hanya ada satu *thread* yang memiliki *state* **Running**. Ketika pada suatu saat tidak ada *thread* yang masuk ke *state* **Running**, maka *thread* lain yang sedang berada pada *state* **Runnable** akan segera diubah *state*-nya menjadi **Running** oleh *scheduler*.

Pada Gambar 7.2 juga diperlihatkan bahwa ketika Thread 1 mengalami *delay* atau *state*-nya berubah menjadi **Blocked**, Thread 2 akan berstatus **Running**. Ketika Thread 2 masuk ke fase *delay*, Thread 1 akan masuk ke dalam *state* **Running**.

7.4 *Method* Penting pada Thread

`sleep()`
Method ini akan menghentikan proses pada sebuah *thread* selama beberapa detik. Thread mempunyai 2 buah *method* `sleep` dengan argumen-argumen berikut :

- `sleep(long milis)` merepresentasikan lamanya penghentian proses dilakukan, selama *milis* milidetik.

- `sleep(long milis, int nanos)` yang merepresentasikan lamanya penghentian proses dilakukan, selama *milis* milidetik ditambah *nanos* nanodetik.

Seperti telah dijelaskan, `sleep()` menyebabkan sebuah *thread* masuk ke dalam *state* **Blocked**. Sebagai catatann bahwa *method* `sleep()` ini merupakan *method* statik dan penggunaannya harus berada dalam blok `try...catch` dengan *exception handler* berupa `InterruptedException` atau superclassnya.

join()
Method ini akan mengakibatkan *method* pemanggil akan menunggu sampai `Thread` yang *method* `join()`-nya dipanggil masuk ke dalam *state* **Die**.

currentThread()
Method ini merupakan *method* statik yang menghasilkan obyek `Thread` yang sedang berjalan.

yield()
Method statik ini menyebabkan *runtime* mengalihkan konteks dari *thread* yang sedang berjalan ke *thread* lain yang tersedia yang siap dijalankan. Penggunaan *method* ini merupakan suatu cara untuk memastikan bahwa *thread* dengan prioritas lebih rendah tidak memaksakan untuk berjalan.

start()
Method ini memberitahu *runtime* Java untuk menciptakan konteks *thread* dan menjalankannya. Selanjutnya *method* `run()` pada tujuan *thread* ini akan dipanggil dalam konteks *thread* baru. *Programmer* harus berhati-hati supaya tidak memanggil *method* `start()` lebih dari sekali pada obyek `Thread` yang diberikan.

run()
Method ini adalah badan *thread* yang berjalan. Ini merupakan *method* tunggal dalam *interface* `Runnable`. *Method* `run()` dipanggil dengan *method* `start()` setelah *thread* diinisialisasikan dengan benar. Ketika *method* `run()` selesai, *thread* yang berlaku akan dihentikan.

stop()
Method ini menyebabkan *thread* segera berhenti. Ini sering dijadikan cara tercepat untuk mengakhiri suatu *thread* khususnya jika *method* ini dieksekusi pada *thread* yang berlaku. Pada kasus ini, baris program setelah pemanggilan *method* `stop()` tidak pernah dieksekusi karena konteks jalinan musnah sebelum `stop()` selesai.

suspend()
Method suspend() berbeda dengan stop(). *Method* ini mengambil *thread* tertentu dan menyebabkannya berhenti tanpa menghancurkan *thread* di bawahnya atau keadaan *thread* yang berjalan sebelumnya. Jika suatu *thread* di-*suspend,* panggil *method* resume() pada *thread* yang sama untuk menjalankannya lagi.

resume()
Method ini digunakan untuk menghidupkan *method* yang di-*suspend.* Tidak ada jaminan bahwa *thread* yang dijalankan kembali akan berjalan dengan baik karena mungkin sudah ada *thread* dengan prioritas lebih tinggi yang berjalan. Tetapi, resume() memungkinkan *thread* untuk berjalan kembali.

setPriority()
Method ini mengisi prioritas suatu *thread* dengan beseran *integer* yang dimasukkan. Ada sejumlah konstanta prioritas yang telah ditetapkan dalam kelas Thread, yaitu MIN_PRIORITY, NORM_PRIORITY, dan MAX_PRIORITY yang secara berurut bernilai 1, 5, dan 10.

getPriority()
Method ini menghasilkan prioritas *thread* saat itu berupa suatu nilai antara 1 dan 10.

7.5 Sinkronisasi Thread

Beberapa *thread* dalam suatu aplikasi dapat mengakses atribut pada satu obyek yang sama. Dan seringkali dalam disain atribut pada obyek yang sama atau obyek yang di-*share* tersebut memiliki aturan dalam penentuan nilainya. Sebagai contoh, pada aplikasi sebuah sistem pemesanan tiket kereta api yang kita sebut saja Ticket Box. Ada 2 atribut yang penentuan nilainya memiliki aturan, yaitu :

a. Persediaan tiket yang memiliki aturan :

- Nilainya tidak boleh lebih rendah dari 0
- Nilainya tidak boleh lebih tinggi dari kapasitas maksimum kursi
- Jika nilainya mencapai kapasitas maksimum kursi (belum ada tiket yang dipesan), maka proses pembatalan tidak akan menghasilkan updating nilai.
- Jika nilainya berada pada angka 0 (semua tiket terbeli), maka proses pemesanan tidak akan menghasilkan updating nilai.

b. Income yang dihasilkan dari penjualan tiket yang memiliki aturan :

- Nilainya tidak boleh lebih rendah dari 0
- Jika petugas loket melakukan proses pemesanan dan nilai persediaan tiket ter-*update*, maka nilai income harus ditambah sejumlah harga satuan yang ditetapkan.

- Jika petugas loket melakukan proses pembatalan dan nilai persediaan tiket ter-*update*, maka nilai income harus dikurangi sejumlah harga satuan atau jumlah uang yang dikembalikan.
- Jika tidak terjadi *updating* nilai persediaan tiket, maka tidak terjadi juga updating nilai income.

Misalnya pada sebuah pusat reservasi tiket, terdapat 3 loket, yaitu Loket 1, Loket 2, dan Loket 3. Jika seorang petugas di Loket 1 akan melakukan proses pemesanan/pembatalan, maka langkah yang dilakukan adalah (jika semua persyaratan terpenuhi untuk update) :

1. Updating nilai persediaan :
   jika prosesnya adalah pemesanan :
   jika persediaan > 0 :
   persediaan = persediaan - 1
   updating income diizinkan.

   Jika persediaan <=0 :
   tidak dilakukan updating persediaan
   updating income tidak diizinkan.

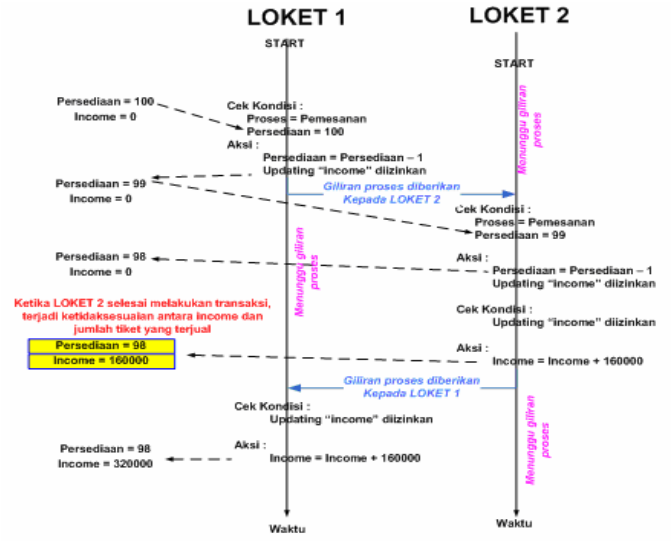
   jika prosesnya adalah pembatalan :
   jika persediaan < jumlah tiket maksimum :
   persediaan = persediaan + 1
   updating income diizinkan

   jika persediaan >= jumlah tiket maksimum :
   tidak dilakukan updating persediaan
   updating income tidak diizinkan.

2. Updating nilai income :
   jika updating income diizinkan :
   jika prosesnya adalah pemesanan :
   income = income + harga satuan
   jika prosesnya pembatalan :
   income = income - harga satuan

Sebuah proses pemesanan / pembatalan akan dikatakan sebagai proses yang lengkap, apabila kedua langkah tersebut telah selesai dilaksanakan.

Jika program bekerja dalam *single thread* (misalnya hanya terdapat 1 loket saja), maka hal ini tidak menimbulkan masalah. Masalah akan muncul ketika program bekerja dalam *multiple thread* (dengan menggunakan lebih dari 1 loket yang berjalan paralel). Dengan asumsi semua loket mengakses ticket box yang sama, maka dapat terjadi kejadian berikut : ketika loket 1 baru saja melakukan updating persediaan tiket (langkah 1), loket 2 masuk ke langkah 1 dan melakukan updating persediaan tiket.



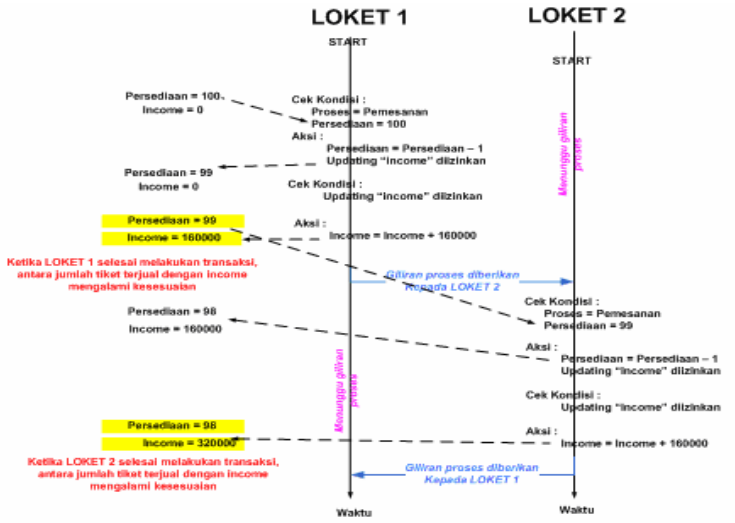
Gambar 7.3 Multi-threading antara 2 loket yang tidak menerapkan sinkronisasi.

Gambar 7.3 menunjukkan proses yang terjadi pada dua buat loket yang diposisikan sebagai Thread. Pada awal proses, Loket 1 lebih dulu melakukan proses (masuk ke *state* **`Running`**), sedangkan Loket 2, yang mulai lebih lambat daripada Loket 1, harus menunggu (masuk ke *state* **`Runnable`**), sampai Loket 1 masuk ke fase menunggu (masuk ke *state* **`Blocking`**).

Ketika Loket 1 melakukan proses, Loket 1 menetapkan bahwa prosesnya adalah PEMESANAN. Oleh karena itu, Loket 1 melakukan pemeriksaan jumlah persediaan tiket. Karena persediaan tiket = 100, yang berarti masih dapat dipesan, maka Loket 1 melakukan aksi, yaitu pengurangan jumlah persediaan dengan 1. Persediaan tiket menjadi 99.Tetapi, ketika proses pada Loket 1 belum selesai (Loket 1 belum melakukan updating income), proses berpindah ke Loket 2, yang melakukan proses PEMESANAN. Loket 2 akan melakukan pemeriksaan jumlah persediaan. Karena jumlah persediaan adalah 99, maka proses pemesanan dapat dilakukan. Loket 2 akan melakukan aksi, berupa pengurangan jumlah persediaan menjadi 98 dan menyatakan bahwa *updating* income diizinkan. Pada Gambar 7.3 diasumsikan Loket 1 masih berada pada *state* **`Blocking`** sehingga Loket 2 masih dapat melakukan prosesnya.

Loket 2 akan melakukan pemeriksaan, apakah *income* dapat di-*update*. Karena pada aksi sebelumnya telah dinyatakan bahwa *updating* income dapat dilakukan, maka langkah berikutnya adalah melakukan aksi, yaitu menjumlahkan income dengan harga satuan, yang diasumsikan = 160000.

Perhatikan komposisi persediaan-income yang diberi warna kuning pada Gambar 7.4. Terlihat suatu ketidak-konsistenan setelah Loket 2 menyelesaikan transaksi. Persediaan tercatat 98, yang berarti 2 tiket telah terjual, tetapi *income* yang diterima adalah 160000, yang setara dengan 1 tiket terjual. Hal ini dikarenakan ketika transaksi pada Loket 1 belum selesai, Loket 2 mendapat giliran melakukan transaksi. Akibatnya, Loket 2 memulai transaksi dengan kondisi awal persediaan dan *income* yang tidak konsisten. Akhirnya, setelah Loket 2 selesai bertransaksi, hasil akhir persediaan - income juga tidak konsisten. Untuk mencegah terjadinya ketidak-konsistenan semacam ini, perlu dilakukan **sinkronisasi *thread*.**



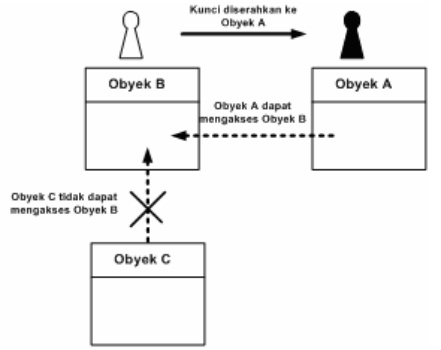
Gambar 7.4 Multithreading antara 2 loket yang menerapkan Sinkronisasi.

Contoh sinkronisasi *thread* dapat dilihat pada Gambar 7.4. Pada Gambar 7.4 diperlihatkan bahwa sebelum Loket 1 menyelesaikan transaksinya, Loket 2 akan terus menunggu. Dengan disain seperti ini, dijamin nilai-nilai pada persediaan dan *income* akan konsisten.

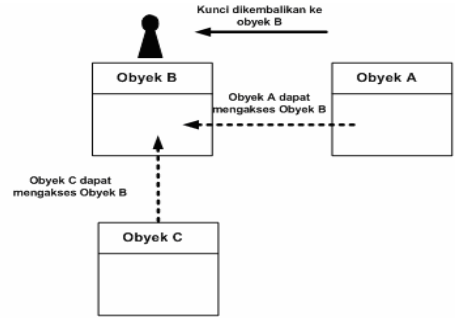
7.6 Konsep Lock pada Obyek dan Penggunaan Kata Kunci *synchronized*

7.6.1 Konsep *lock* pada obyek

Untuk dapat melakukan sinkronisasi *thread*, teknologi Java memperkenalkan terminologi *lock* pada obyek. Lock merupakan ”kunci” virtual yang dimiliki oleh obyek. Kunci virtual ini dapat dimiliki oleh obyek lain yang mengakses obyek pemilik kunci. Akibatnya adalah : selama obyek A memiliki kunci dari obyek yang diakses (misalnya obyek B) , maka obyek C tidak dapat mengakses obyek B. Obyek C akan terus menunggu sampai obyek A melepaskan kunci obyek B dan obyek B memiliki kembali kunci tersebut. Untuk lebih jelasnya, perhatikan Gambar 7.5 dan Gambar 7.6.



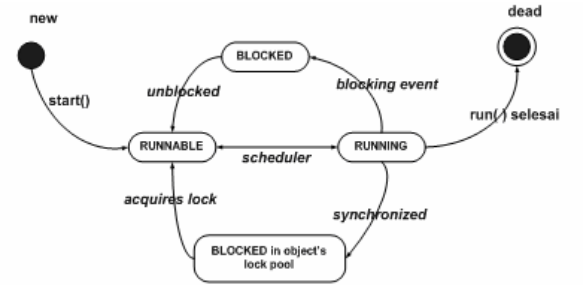
Gambar 7.5 Lock obyek B dipegang oleh obyek A



Gambar 7.6 Lock obyek B dikembalikan oleh obyek A

7.6.2 Kata Kunci *synchronized*

Untuk menghasilkan sinkronisasi di antara *thread*, maka teknologi Java menyediakan kata kunci *synchronized* yang akan menyebabkan lock / kunci obyek berpindah ke thread yang mengakses obyek tersebut.



Gambar 7.7 Diagram state Model II

Penggunaan kata kunci *synchronized* menyebabkan diagram state pada model I dimodifikasi menjadi model II, seperti pada Gambar 7.7.

Pada diagram state model II, terdapat tambahan *state*, yaitu **`BLOCKED in object's lock pool`**. Artinya, ketika ada *thread* yang mengakses suatu obyek B yang memiliki *method* atau blok yang menggunakan kata kunci *synchronized*, *thread* lain harus menunggu untuk mengakses obyek B tersebut. *Thread* lain tersebut masuk ke dalam *state* **`BLOCKED in object's lock pool`**.

Ketika *thread* pemegang *lock* tersebut mengembalikan kunci kepada obyek B, maka *thread* yang sedang menunggu di **`BLOCKED in object's lock pool`** mendapatkan kunci obyek B, dan masuk ke dalam *state* **`Runnable`**. Seperti pada Model I, *thread* pemegang kunci obyek B tersebut dipilih oleh *scheduler* untuk masuk ke *state* **`Running`**. Dengan demikian, perilaku yang dihasilkan adalah sebagai berikut :

- Ketika sebuah *method* / blok dari sebuah obyek yang menggunakan kata kunci *synchronized* diakses oleh sebuah *thread*, *thread* lain tidak dapat mengakses *method-method* milik obyek tersebut.
- Ketika sebuah *method* / blok dari sebuah obyek yang menggunakan kata kunci *synchronized* diakses oleh sebuah *thread*, thread lain masih dapat mengakses *method-method* yang tidak tersinkronisasi milik obyek tersebut.

7.6.2.1 Sinkronisasi pada *method*

Sinkronisasi pada *method* berarti sinkronisasi dilakukan untuk keseluruhan isi *method*. Akibatnya, ketika *method* tersebut diakses oleh sebuah *thread*, maka *thread* lain harus menunggu sampai *method* tersebut selesai dieksekusi.

Untuk mendefinisikan sebuah *method* tersinkronisasi pada kelas, *syntax*-nya adalah sebagai berikut :

```
[modifiers] synchronized data_type identifier(arguments) {
        //code block
}
```

Contoh penggunaan kata kunci *synchronized* pada method adalah sebagai berikut :

```
public synchronized void pesan(String nama) {
        //code block
}
```

untuk lebih jelasnya, perhatikan contoh kode program berikut ini.

```
class Callme{
        synchronized void call(String msg){
                System.out.print("[" + msg);
                try {
                        Thread.sleep(1000);
                }catch (Exception e){}
                System.out.println("]");
        }
}

class Caller implements Runnable{
        String msg;
        Callme target;

        public Caller(Callme t, String s){
                target = t;
                msg = s;
                new Thread(this).start();
        }

        public void run(){
                target.call(msg);
        }
}
```

```
public class Synchronizer{
	public static void main (String[] args){
		Callme target = new Callme();
		new Caller(target, "Java");
		new Caller(target, "Competency");
		new Caller(target, "Center ITB");
	}
}
```

Pada contoh program di atas terdapat tiga kelas sederhana. Kelas pertama adalah kelas `Callme` yang memiliki *method* tunggal bernama `call(String msg)`. *Method* ini memerlukan parameter `String msg`. *Method* ini mencoba mencetak `String msg` di antara tanda kurung siku. Perhatikan bahwa setelah mencetak kurung siku buka dan `string msg`, *method* ini akan memanggil `Thread.sleep(1000)` yang menghentikan *thread* yang sedang berjalan selama satu detik.

Kelas kedua adalah kelas `Caller`. Kelas ini memiliki konstruktor yang memerlukan referensi ke suatu instan kelas `Callme` dan suatu `String` yang secara berurut disimpan dalam variabel `target` dan `msg`. Konstruktor juga membuat suatu `Thread` baru yang akan memanggil *method* `run()` milik obyek ini. Selanjutnya, *thread* segera dimulai. *Method* `run()` pada `Caller` memanggil *method* `call` milik obyek `target` (instan kelas `Callme`) dengan melewatkan `string msg`.

Akhirnya, kelas `Shyncronizer` membuat instan tunggal dari kelas `Callme` dan tiga instan kelas `Caller` masing-masing dengan `String` pesan yang berbeda. Instan kelas `Callme` yang sama dilewatkan pada setiap `Caller`. Keluaran dari program di atas adalah sebagai berikut :

```
[Java]
[Competency]
[Center ITB]
```

Apa yang akan terjadi, jika *method* `call` milik kelas `Callme` tidak disertakan keyword `synchronized` ?

7.6.2.2 Sinkronisasi pada baris program tertentu

Terkadang tidak semua baris pada *method* perlu disinkronisasi. Mungkin hanya satu atau dua baris pada *method* yang dibutuhkan untuk sinkronisasi. Agar bagian dari *method* saja yang disinkronisasi, maka *syntax*-nya adalah sebagai berikut :

```
[modifiers] data_type identifier(arguments) {
        //code block 1
        synchronized(Object obj){
                //code block 2
        }
        //code block 3
}
```

Contoh penggunaan kata kunci *syncronized* pada satu blok baris program pada method adalah sebagai berikut :

```
public synchronized void batal(String nama){
   if(persediaan<MAX){
      synchronized(this){
         this.persediaan++;
         updateOK=true;
      }
      System.out.println(nama + " Pembatalan NOTIFY");
      notifyAll();
   }
   // baris program selanjutnya
}
```

Pada contoh di atas, ketika sebuah Thread I memanggil *method* batal(), Thread II masih punya kesempatan untuk mengakses *method* batal() bersamaan dengan Thread I. Tetapi ketika Thread I mengakses blok *synchronized*, maka ketika Thread II selesai memeriksa kondisi persediaan < MAX, Thread II akan menunggu sampai Thread I selesai mengeksekusi baris-baris program pada blok *synchronized*.

Perhatikan bahwa blok *synchronized* menggunakan argumen sebuah obyek. Artinya, ketika sebuah *thread* mengakses blok *synchronized*, maka *thread* lain tidak dapat mengakses *method-me*thod / baris-baris program tersinkronisasi pada obyek yang menjadi argumen *synchronized*.

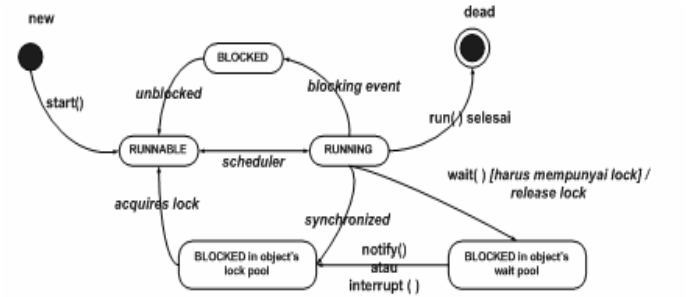
7.7 Komunikasi Antar *Thread*

Sebuah *thread* dapat melakukan komunikasi dengan *thread* lainnya. Komunikasi tersebut berkaitan dengan penggiliran pemakaian kunci obyek. Ketika sebuah *thread* selesai mengakses sebuah *method* / blok tersinkronisasi dari sebuah *thread*, obyek tersebut akan mengembalikan kunci obyek ke obyek pemiliknya.

Secara otomatis ketika sebuah kunci obyek dikembalikan ke obyek bersangkutan, maka salah satu dari *thread-thread* yang menunggu

giliran mendapatkan kunci obyek (**BLOCKED in object's lock pool**) akan mendapatkan kunci tersebut dan masuk ke dalam *state* **Runnable**. Ini yang disebut komunikasi antar *thread*.

Komunikasi antar *thread* dapat dibuat tidak otomatis, yaitu dengan menggunakan *method* `wait()` , `notify()` , dan `notifyAll()` milik obyek yang kuncinya dipegang oleh *thread*. Dengan penggunaan ketiga *method* tersebut, model diagram state pada *thread* berkembang menjadi model III seperti pada Gambar 7.8.



Gambar 7.8 DiagramState Model III.

Pada diagram state model III, terdapat satu tambahan *state*, yaitu **BLOCKED in object's wait pool**. *State* ini merupakan *state* untuk obyek-obyek yang menunggu kunci obyek dengan cara memanggil method `wait()`.

Method `wait()` menyebabkan *thread* terakhir yang mengakses obyek melepaskan kunci obyek dan menunggu sampai *thread* lain yang mengambil kunci obyek tersebut melepaskannya kembali ke obyek. Terdapat 2 jenis method `wait()`, yaitu :

a. `wait()`menunggu sampai waktu tak berhingga sampai ada *thread* yang menyerahkan kunci obyek kepada obyek bersangkutan. Jika ada obyek yang menyerahkan kunci obyek kepada obyek bersangkutan, maka method `notify()` akan dipanggil dari *thread* yang berada dalam *state* **BLOCKED in object's wait pool**, sehingga *thread* pindah ke *state* **BLOCKED in object's lock pool**.
b. `wait ( long timeout )` menunggu sampai *timeout* milidetik. Jika sampai batas waktu *timeout* tidak ada obyek yang menyerahkan kunci obyek, maka secara otomatis *method* `interrupt()` dari *thread* akan dipanggil, dan *thread* masuk ke dalam **state BLOCKED in object's lock pool**.

Sebagai catatan bahwa method `wait()` hanya dapat dipanggil oleh *thread* pemegang kunci obyek. Jika sebuah *thread* yang tidak memegang kunci obyek memanggil `wait()`, maka akan terjadi eksepsi, yaitu `IllegalMonitorStateException`.

Thread yang berada pada *state* **BLOCKED in object's wait pool** tidak dapat menerima kunci obyek, meskipun kunci obyek telah dikembalikan ke obyek bersangkutan. Untuk dapat menerima kunci obyek, *thread* harus pindah ke state **BLOCKED in object's lock pool** terlebih dahulu dengan menggunakan `notify()`, `notifyAll()` , atau `interrupt()`.

Method `notify()` akan menyebabkan *thread* pemegang kunci obyek melepaskan kunci obyek dan obyek pemilik kunci memilih salah satu *thread* pada *state* **BLOCKED in object's wait pool** untuk diubah *state*-nya ke **BLOCKED in object's lock pool**.

Method `notifyAll()` akan menyebabkan *thread* pemegang kunci obyek melepaskan kunci obyek dan obyek pemilik kunci memindahkan semua *thread* yang berada pada *state* **BLOCK in object's wait pool** ke *state* **BLOCKED in object's lock pool**. Untuk lebih jelasnya, perhatikan kode program berikut ini.

```
class Transaction{
        int i;
        synchronized int get(){
                System.out.println("Got : " + i);
                return i;
        }

        synchronized void put(int i){
                this.i = i;
                System.out.println("Put : " + i);
        }
}

class Producer implements Runnable{
        Transaction trans;

        Producer(Transaction trans){
                this.trans = trans;
                new Thread(this, "Producer").start();
        }

        public void run(){
                int i = 0;

                while(true){
                        trans.put(i++);
                }
        }
}
```

```
class Customer implements Runnable {
        Transaction trans;

        Customer(Transaction trans){
                this.trans = trans;
                new Thread(this, "Customer").start();
        }

        public void run(){
                while(true){
                        trans.get();
                }
        }
}

public class TransactionTest{
        public static void main (String args[]){
                Transaction trans = new Transaction();
                new Producer(trans);
                new Customer(trans);
        }
}
```

Contoh program di atas terdiri dari empat kelas sederhana, yaitu `Transaction`, `Producer`, `Customer`, dan `TransactionTest`. Kelas `Transaction` mendefinisikan antrian yang aksesnya ingin disinkronkan. Kelas `Producer` menghasilkan isi antrian. Kelas `Customer` menggunakan isi antrian. Kelas `TransactionTest` menghasilkan kelas `Transaction`, `Producer`, dan `Customer` tunggal.

Meskipun *method* `put` dan `get` pada `Transaction` sudah disinkronkan, masih tidak mungkin menghentikan produsen supaya tidak mendahului konsumen. Selain itu, juga tidak mungkin menghentikan konsumen supaya tidak mengkonsumsi nilai antrian yang sama lebih dari satu kali. Jadi, program di atas akan menghasilkan keluaran yang masih salah seperti berikut ini.

```
Put : 1
Got : 1
Got : 1
Got : 1
Got : 1
Got : 1
Put : 2
Put : 3
Put : 4
Put : 5
Put : 6
Put : 7
Put : 7
```

Cara untuk mengatasi masalah di atas adalah dengan menggunakan *method* wait() dan notify() untuk memberi isyarat dua arah. *Method* get terdapat mekanisme untuk menunggu (*wait*) sampai Producer memberitahu bahwa sudah ada data yang siap. Setelah Customer memanggil data tersebut, Customer memberitahu (*notify*) Producer bahwa sudah boleh menambah data di antrian. *Method* put terdapat mekanisme menunggu (*wait*) sampai Customer sudah menghilangkan data terakhir yang disimpan di antrian. Lalu, setelah memasukkan data baru, Produser memberitahu (*notify*) Customer supaya menghilangkan data baru tersebut. Berikut kelas Transaction yang sudah dimodifikasi.

```
class Transaction{
   int i;
   boolean valueSet = true;

   synchronized int get(){
          if (!valueSet)
                  try {
                     wait();
                  }catch(InterruptedException e){}
          System.out.println("Got :" + i);
          valueSet = false;
          notify();
          return i;
   }

   synchronized void put(int i){
          if (valueSet)
                  try {
                     wait();
                  }catch(InterruptedException e){}
          this.i = i;
          valueSet = true;
          System.out.println("Put :" + i);
          notify();
   }
}
```

Keluaran program setelah kelas Transaction dimodifikasi adalah sebagai berikut.

```
Put : 1
Got : 1
Put : 2
Got : 2
Put : 3
Got : 3
Put : 4
Got : 4
Put : 5
Got : 5
```

7.8 *DeadLock*

Deadlock biasanya jarang terjadi, tetapi sangat sulit untuk menelusuri keadaan kesalahan di mana dua jalinan memiliki saling kebergantungan pada sepasang obyek yang disinkronkan. Contohnya, satu *thread* memasuki monitor pada obyek X dan *thread* lain memasuki monitor pada Obyek Y. Kemudian jika X mencoba memanggil suatu *method* yang disinkronkan Y, maka X akan membloknya. Tetapi, jika Y pada gilirannya mencoba memanggil suatu *method* yang disinkronkan pada X, maka Y akan menunggu selamanya karena untuk mendapatkan kunci pada X, pertama kali X harus melepaskan kuncinya pada Y sehingga *thread* pertama dapat dilengkapi. Untuk lebih jelasnya, perhatikan contoh program berikut ini.

```
class X {
     synchronized void foo(Y y){
        String name = Thread.currentThread().getName();
        System.out.println(name + " memasuki X.foo");
        try{
             Thread.sleep(1000);
        }catch (Exception e){}
        System.out.println(name + " mencoba memanggil
                           Y.last()");
        y.last();
     }

     synchronized void last(){
        System.out.println(" Di dalam X.last()");
     }
}


class Y {
     synchronized void bar(X x){
        String name = Thread.currentThread().getName();
        System.out.println(name + " memasuki Y.bar");
        try{
            Thread.sleep(1000);
        }catch (Exception e){}
        System.out.println(name + " mencoba memanggil
                           X.last()");
        x.last();
     }

     synchronized void last(){
        System.out.println(" Di dalam X.last()");
     }
}
```

```
class A implements Runnable{
        X x = new X();
        Y y = new Y();

      A(){
          Thread.currentThread().setName("ITB");
          new Thread(this, "JCC ITB").start();
          x.foo(y);
          System.out.println("Kembali ke Thread utama");
      }

      public void run(){
          y.bar(x);
          System.out.println("Kembali ke Thread lain");
      }

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

Pada contoh di atas dibuatkan dua kelas, yaitu `A` dan `B` yang secara berurut memiliki *method* `foo` dan `bar`, yang berhenti sebentar sebelum mencoba memanggil satu *method* dalam masing-masing *method* lain. Kelas utama bernama `A` menciptakan instan `A` dan `B`, lalu mengambil *thread* kedua untuk menyiapkan keadaan *deadlock*. *Method* `foo` dan `bar` menggunakan `sleep` sebagai cara untuk memaksa munculnya keadaan *deadlock*.

`JCC ITB` memiliki monitor pada `y` sambil menunggu monitor pada `x`. pada saat yang sama `ITB` memiliki `x` dan menunggu untuk mengambil `b`. Program ini tidak akan pernah selesai. Berikut keluaran dari program di atas.

```
ITB memasuki X.foo
JCC ITB memasuki Y.bar
ITB mencoba memanggil Y.last()
JCC ITB mencoba memanggil X.last()
```

8 *INPUT / OUTPUT* (I/O)

8.1 Java I/O

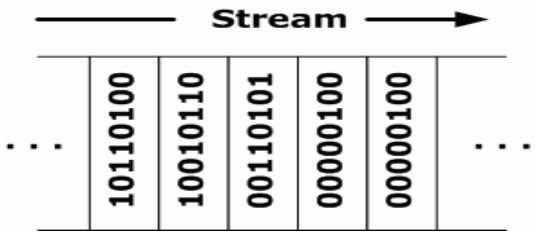
Java I/O (*Java Input/Output*) adalah fitur dalam teknologi Java yang mengakomodasi perpindahan data dari satu sumber data (*data source*) ke tujuan (*data sink*) seperti perpindahan satu karakter yang diketikkan melalui *keyboard* ke layar monitor sehingga ketika satu huruf diketikkan oleh pengguna melalui *keyboard,* maka beberapa saat kemudian huruf tersebut ditampilkan pada layar monitor.

Fitur I/O pada teknologi Java terdapat pada package `java.io`. Beberapa kelas dan *interface* yang termasuk dalam package `java.io` adalah :

a. `DataInput`
b. `DataOutput`
c. `BufferedInputStream`
d. `BufferedOutputStream`
e. `FileReader`
f. `FileWriter`
g. `FileInputStream`
h. `FileOutputStream`

8.2 *Stream*

Untuk memahami Java I/O, perlu dipahami konsep *stream. Stream* adalah sebuah baris berurut dari *byte-byte* data dengan panjang yang tidak tertentu (*undeterministic*). Perhatikan ilustrasi *stream* pada Gambar berikut ini.



Gambar 8.1 Ilustrasi suatu *stream*.

Informasi yang dibawa oleh *stream* merupakan *byte-byte* urutan kombinasi bit-bit 1 dan 0 dan belum mengalami interpretasi tertentu, misalnya interpretasi karakter *Unicode*.

Pada teknologi Java, *stream* terbagi atas 2 kategori, yaitu :

a. *Byte stream*, yaitu *stream* yang membawa informasi berupa *byte-byte* yang dipresentasikan sebagai bilangan-bilangan kode ASCII.
b. *Character stream*, yaitu *stream* yang membawa informasi berupa *byte-byte* yang direpresentasikan sebagai karakter.

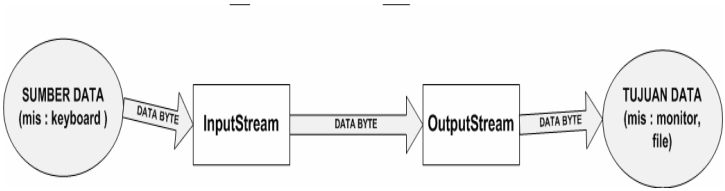
Proses aliran data pada teknologi Java menggunakan terminologi "sumber data" dan "tujuan data" di mana arah aliran data adalah dari "sumber data" menuju ke "tujuan data". Sumber data (*data source*) adalah perangkat/entitas yang menyimpan data yang akan diambil oleh program untuk diproses. Sedangkan tujuan data (*data sink*) adalah perangkat/entitas yang berfungsi sebagai tempat menyimpan/menampilkan data yang telah selesai diproses oleh program.

Konsep sumber data dan tujuan data ini membutuhkan obyek-obyek yang dapat membaca, menampung, dan menuliskan data. Oleh karena itu teknologi Java memperkenalkan kelas `InputStream`, `OutputStream`, `Reader`, dan `Writer` yang masing-masing menangani pembacaan informasi pada *byte stream*, penulisan informasi pada *byte stream*, pembacaan informasi pada *character stream*, dan penulisan informasi pada *character stream*.

Sumber data dan tujuan data lebih umum disebut *node stream*. Sedangkan aliran data lebih umum disebut *stream* saja.

8.2.1 *Byte Stream*

Byte Stream adalah *stream* yang memuat informasi berbasis *byte*. Informasi ini akan dibaca sebagai *byte-byte*, dan tidak dilakukan konversi ke karakter. Jadi meskipun pada *Byte Stream* terdapat komposisi byte yang dapat direpresentasikan ke dalam karakter, byte tersebut tidak dikonversi ke dalam karakter.



Gambar 8.2 Ilustrasi *Byte Stream*.

8.2.1.1 InputStream

InputStream adalah sebuah kelas abstrak yang merepresentasikan sebuah *input stream* yang memuat informasi berupa barisan *byte*. InputStream akan membaca *byte-byte* dari sumber data, misalnya *keyboard*.

InputStream memiliki beberapa *method* sebagai berikut :

a. `public abstract int read() throws IOException` : membaca *byte* berikutnya dari *input stream*. *Method* ini akan menghentikan proses sampai terjadi pemasukan data dari sumber data.

b. `public int read (byte [] b ) throws IOException` : membaca beberapa bytes data dan menyimpannya dalam *buffer b.*

c. `public int read (byte[] data, int offset, int length) throws IOException` : membaca beberapa bytes data dan menyimpannya dalam *buffer b*, dimulai dari indeks ke-*offset* sepanjang *length*.

d. `public int available () throws IOException` : menghasilkan jumlah bytes yang dapat dibaca.

e. `public void close() throws IOException` : menutup *input stream* dan membebaskan *resource* sistem yang berasosiasi dengan *stream*.

f. `public synchronized void mark(int readlimit) throws IOException` : menandai posisi terakhir pada *input stream*. Argumen *readlimit* menunjukkan posisi *mark* di mana *byte* yang dibaca sebelum posisi *mark* tersebut dianggap *invalidate*.

Perhatikan bahwa *method-method* pada InputStream didefinisikan dengan memberikan eksepsi, yaitu IOException. Ini berarti pemanggilan *method-method* tersebut harus dilakukan di dalam blok try... catch dengan instans dari IOException atau kelas induknya sebagai obyek yang menangani eksepsi.

Pemanggilan *method* pada InputStream diperlihatkan pada contoh program di bawah ini. Sebagai catatan, obyek yang digunakan adalah instans dari kelas DataInputStream yang merupakan turunan dari kelas InputStream. Pemanggilan kelas turunan ini disebabkan oleh *modifier* abstract pada kelas InputStream yang berarti tidak dapat dibuat sebuah obyek/instans dari kelas InputStream. Agar instans InputStream dapat dibuat, harus dilakukan instanstiasi dari kelas turunannya dalam hal ini DataInputStream.

```
import java.io.*;

public class InputOutput1{
   public static void main(String[] args){
      DataInputStream dis;
      int hasil;
      try{
         dis = new DataInputStream(System.in);
         while((hasil=dis.read())!=-1) {
            System.out.println("ASCII = "+ hasil);
         }
         dis.close();
      } catch(IOException e){}
   }
}
```

Contoh di atas menunjukkan sebuah program untuk membaca data yang dimasukkan melalui *keyboard*. Pada *method* main(), dideklarasikan obyek dis yang merupakan instans dari kelas DataInputStream dan variabel primitif hasil bertipe int. Pada blok *try...catch* obyek dis diinstanstiasi dengan menggunakan konstruktor DataInputStream(InputStream). Pada kasus ini argumen yang digunakan sebagai inputstream adalah System.in.

Sebagai catatan, variabel statik in pada kelas System merupakan inputstream standar pada teknologi Java yang menyimpan data yang dikirimkan melalui *keyboard*. Dengan demikian, obyek dis akan membaca semua informasi yang masuk melalui *keyboard*.

Pada baris ke-9 dilakukan pembacaan data yang dimasukkan melalui *keyboard*. Program akan menunggu sampai terjadi penekanan tombol *Enter* pada *keyboard*, kemudian data tersebut dikembalikan ke variabel hasil. Data yang dimasukkan ke hasil adalah kode ASCII dari karakter *keyboard* yang dimasukkan.

Jika kita mengetikkan **abc** pada *keyboard*, maka output yang dihasilkan oleh program di atas adalah sebagai berikut :

```
ASCII = 97
```

```
ASCII = 98
ASCII = 99
ASCII = 13
ASCII = 10
```

Nilai ASCII “97” melambangkan huruf “a”, nilai “98” melambangkan huruf “b”, nilai ”99” melambangkan huruf ”c”, nilai “13” melambangkan “carriage return”, dan nilai “10” melambangkan “linefeed”. Perlu dicatat bahwa setiap penekanan tombol *Enter* pada *keyboard*, data yang dikirimkan oleh *keyboard* akan diakhiri dengan “carriage return” diikuti dengan “linefeed”. Konversi karakter pada *keyboard* ke kode ASCII dapat dilihat pada Tabel 8.1.

Pada baris ke-10 nilai yang disimpan oleh variabel `hasil` akan dicetak ke layar monitor. Nilai yang dicetak adalah kode ASCII dari karakter yang bersesuaian. Langkah terakhir adalah menutup *input stream* `dis`. Penutupan *stream* ini sebaiknya wajib dilakukan karena JVM memiliki kemampuan terbatas untuk jumlah *stream* maksimum yang dibuka. Meskipun dalam contoh program di atas tidak terlihat keterbatasan JVM tersebut, alangkah baiknya jika penutupan *stream* selalu dilakukan untuk menghemat *resource*.

Tabel 8.1
Set Karakter ASCII

Code	Character	Code	Character	Code	Character	Code	Character
0	nul (null)	32	space	64	@	96	`
1	soh (start of header)	33	!	65	A	97	a
2	stx (start of text)	34	"	66	B	98	b
3	etx (end of text)	35	#	67	C	99	c
4	eot (end of transmission)	36	$	68	D	100	d
5	enq (enquiry)	37	%	69	E	101	e
6	ack (acknowledge)	38	&	70	F	102	f
7	bel (bell)	39	'	71	G	103	g
8	bs (backspace)	40	(	72	H	104	h
9	tab (tab)	41	)	73	I	105	i
10	lf (linefeed)	42	*	74	J	106	j
11	vtb (vertical tab)	43	+	75	K	107	k
12	ff (formfeed)	44	,	76	L	108	l
13	cr (carriage return)	45	-	77	M	109	m
14	so (shift out)	46	.	78	N	110	n
15	si (shift in)	47	/	79	O	111	o
16	dle (data link escape)	48	0	80	P	112	p
17	dc1 (device control 1, XON)	49	1	81	Q	113	q
18	dc2 (device control 2)	50	2	82	R	114	r
19	dc3 (device control 3, XOFF)	51	3	83	S	115	s
20	dc4 (device control 4)	52	4	84	T	116	t
21	nak (negative acknowledge)	53	5	85	U	117	u
22	syn (synchronous idle)	54	6	86	V	118	v
23	etb (end of transmission block)	55	7	87	W	119	w
24	can (cancel)	56	8	88	X	120	x
25	em (end of medium)	57	9	89	Y	121	y
26	sub (substitute)	58	:	90	Z	122	z
27	esc (escape)	59	;	91	[	123	{
28	is4 (file separator)	60	<	92	\	124	|
29	is3 (group separator)	61	=	93	]	125	}
30	is2 (record separator)	62	>	94	^	126	~
31	is1 (unit separator)	63	?	95	_	127	del (delete)

8.2.1.2 OutputStream

`OutputStream` adalah sebuah kelas abstrak yang merepresentasikan sebuah *output stream* yang akan menjadi tujuan aliran urutan *byte*. `OutputStream` akan menampung data yang diberikan kepadanya dan menuliskannya ke media *output* yang bersesuaian, misalnya ke layar monitor atau *file*.

Pada contoh sebelumnya ditunjukkan hasil pembacaan *byte* yang dicetak ke layar monitor. Variabel `System.out` adalah variabel statik yang merupakan instans dari kelas `PrintStream`, kelas turunan dari `OutputStream`. Variabel `System.out` merupakan *output stream* standar yang menuliskan informasi *byte* ke layar monitor.

`OutputStream` memiliki beberapa *method* sebagai berikut :

a. `public void write (byte[] cbuf) throws IOException` : Menuliskan semua isi *array byte* ke tujuan data.

b. `public abstract void write( byte[] cbuf, int off, int len) throws IOException` : Menuliskan sebagian isi *array byte* ke tujuan data dengan parameter `off` adalah *offset* / nomor indeks dimulai pembacaan *array*, sedangkan `len` merepresentasikan panjang *byte* yang akan dibaca.

c. `public void write (int b) throws IOException` : Menuliskan satu byte `b` ke tujuan data

d. `public void flush () throws IOException` : "Mendorong" data yang dituliskan pada *output stream* sehingga benar-benar terkirim ke tujuan data.

Contoh kode program berikut menunjukkan bagaimana informasi yang dimasukkan melalui *keyboard* dicetak ke dalam *file*.

```
import java.io.*;

public class InputOutput2{
   FileOutputStream fos = null;
   BufferedInputStream bis = null;
   File file = null;
   int hasil;

   public static void main(String[] args){
      InputOutput2 io2 = new InputOutput2();
   }
public InputOutput2(){
```

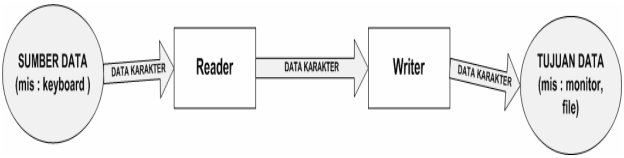
```
        try{
           file = new File("io2.txt");
           fos = new FileOutputStream(file);
           bis = new BufferedInputStream(System.in);
           while((hasil=bis.read())!=-1){
              fos.write(hasil);
             System.out.println(hasil);
           }
           fos.close();
           bis.close();
        }catch(IOException e){
           e.printStackTrace();
        }
     }
  }
```

Pada contoh program di atas dilakukan penyalinan data yang dikirim oleh *keyboard* ke *file* bernama **`io2.txt`**. Data dari *keyboard* ditangani oleh obyek `bis` (`BufferedInputStream`). Program di atas menggunakan method `read()`. Jika method `read()` menghasilkan nilai `-1`, maka pembacaan sudah mencapai akhir *stream*. Selanjutnya program tersebut menampilkan data yang dikirim dari keyboard dan menuliskannya ke *file* **`io2.txt`**.

8.2.2 *Character Stream*

Berbeda dengan *byte stream*, *character stream* mengakomodasi informasi berupa karakter. Artinya, data byte yang ditransfer akan dikonversi menjadi data karakter. Secara fisik sebenarnya data yang ditransfer oleh *character stream* sama saja dengan *byte stream*. Perbedaannya terletak di representasi data yang ditransfer.

Perbedaan lain antara *byte stream* dan *character stream* adalah terminologi yang digunakan untuk menyebut *node stream*. Pada *character stream*, *node stream* yang digunakan untuk menerima data *input* diberi nama dengan format `xxxReader`, dengan `xxx` adalah jenis *node stream input*. Sedangkan *node stream* yang digunakan untuk menuliskan data diberi nama dengan format `xxxWriter`, dengan `xxx` adalah jenis *node stream output*.



Gambar 8.3 Ilustrasi *Character Stream*.

Skema transfer data/informasi pada *character stream* memiliki pola yang sama dengan transfer data pada *byte stream*, seperti pada Gambar 8.3. Data diambil dari sumber data (misalnya *keyboard*/*file*) oleh Reader, dan data dituliskan ke tujuan data (misalnya *monitor*/*file*) oleh Writer.

8.2.2.1. Reader

Reader adalah *node stream* yang berfungsi melakukan pembacaan data karakter dari sumber data. Beberapa *method* pada Reader antara lain :

a. `public int read() throws IOException` : membaca *byte* berikutnya dari *input stream*. *Method* ini akan menghentikan proses sampai terjadi pemasukan data dari sumber data.

b. `public int read (char [] c ) throws IOException` : membaca beberapa *bytes* data dan menyimpannya dalam *buffer c.*

c. `public int read (char[] c, int offset, int length) throws IOException` : membaca beberapa *bytes* data dan menyimpannya dalam *buffer* c, dimulai dari indeks ke-*offset* sepanjang *length*.

d. `public int available ( ) throws IOException` : menghasilkan jumlah *byte* yang dapat dibaca.

e. `public void close() throws IOException` : menutup *input stream* dan membebaskan *resource* sistem yang berasosiasi dengan *stream*.

f. `public void mark(int readlimit) throws IOException` : menandai posisi terakhir pada *input stream*. Argumen *readlimit* menunjukkan posisi *mark* di mana *byte* yang dibaca sebelum posisi mark tersebut dianggap *invalidate*.

Pada Java API, Reader merupakan sebuah kelas abstrak. Untuk dapat menggunakan obyek Reader dalam program, *programmer* dapat melakukan instanstiasi obyek dari turunan kelas Reader. Beberapa kelas turunan dari Reader antara lain :

a. BufferedReader : kelas turunan Reader yang membaca teks dari *character-input-stream* kemudian melakukan *buffering*.

b. InputStreamReader : Kelas turunan Reader yang merupakan jembatan dari *byte stream* ke *character stream*. Kelas ini mempunyai turunan, yaitu FileReader yang membaca karakter dari sebuah *file*.

InputStreamReader biasanya digunakan untuk menerima data *byte* dari sumber data dan langsung mengkonversikannya ke format karakter. *Method* yang digunakan untuk membaca data sama seperti InputStream, yaitu read(). Meskipun fungsinya sebagai konverter dari data byte ke karakter, *method-method* pada InputStreamReader tidak menampilkan nilai karakter dari byte yang dikonversikan.

Tidak adanya *method* yang menampilkan atau mengembalikan nilai karakter mengakibatkan implementasi *reader* ini dalam program selalu dipasangkan dengan sebuah obyek dari kelas BufferedReader. Kelas BufferedReader memiliki *method* yang dapat mengembalikan String, yaitu :

public String readLine () throws IOException	:	Membaca satu baris teks

Penggunaan InputStreamReader ditunjukkan pada contoh kode program berikut.

```
import java.io.*;

public class ReaderExample{
   BufferedReader bufReader = null;
   InputStreamReader isr = null;

   public static void main(String[] args){
        ReaderExample re = new ReaderExample();
   }

   public ReaderExample(){
      try{
         isr = new InputStreamReader(System.in);
         bufReader = new BufferedReader(isr);
         String hasil;
```

```
            while((hasil=bufReader.readLine())!=null){
               System.out.println("Baris yang dibaca = :"
                                  +hasil);
            }

            bufReader.close();
            isr.close();
         }catch(IOException e){}
      }
   }
```

Pada contoh di atas kelas ReaderExample memiliki atribut yaitu obyek BufferedReader bufReader, dan obyek InputStreamReader isr. Pada konstruktor ReaderExample(), terdapat baris-baris kode yang menginstanstiasi atribut-atribut kelas ReaderExample. Atribut isr membaca input dari *keyboard* (System.in). Hasil pembacaan oleh isr akan dimasukkan ke dalam *buffer* yang berada dalam atribut bufReader. Setelah instanstiasi atribut dilakukan *looping* while, yang mempunyai eksperis boolean, untuk mengambil data dari bufReader. *Looping* while akan terus dijalankan sampai pembacaan mencapai akhir dari *stream*. Jika program keluar dari *loop* while, maka atribut-atribut bufReader dan isr ditutup.

Ketika program dijalankan, program akan menunggu masukan dari *keyboard*. Diasumsikan bahwa pengguna memasukkan karakter-karakter “Aku belajar Java”, lalu menekan tombol ENTER. Output yang ditampilkan pada layar monitor adalah sebagai berikut :

```
Baris yang dibaca = :Aku belajar Java
```

Contoh program di atas belum memperlihatkan kemampuan kelas BufferedReader yang dapat membaca data yang terdiri atas beberapa baris. Pada contoh program berikut ditunjukkan cara untuk membaca data dari sebuah *file* dan kemudian menampilkan isi *file* tersebut ke layar monitor.

```
import java.io.*;

public class ReaderFromFileExample{
   BufferedReader bufReader = null;
   FileReader fr = null;

   public static void main(String[] args){
        ReaderFromFileExample re = new
              ReaderFromFileExample();
   }
```

```
    public ReaderFromFileExample(){
       try{
          fr = new FileReader(new File("source.txt"));

          bufReader = new BufferedReader(fr);
          String hasil;

          while((hasil=bufReader.readLine())!=null){
             System.out.println("Baris yang dibaca = :"
                   +hasil);
          }
          bufReader.close();
          fr.close();
       }catch(IOException e){x}
    }
}
```

Pada contoh di atas pembacaan *file* dilakukan menggunakan FileReader (obyek fr) dan BufferedReader (obyek bufReader). Obyek fr membaca data dari file “source.txt”.

Karakter-karakter yang telah dibaca oleh fr akan di-*buffer* oleh bufReader. Skema ini dapat juga dibuat dengan memasukkan *file* ”source.txt” sebagai argumen dari konstruktor FileReader dan memasukkan obyek fr sebagai argumen dari konstruktor BufferedReader.

Pembacaan file ”source.txt” dilakukan dengan menggunakan *method* readLine( ) dari obyek bufReader. Selanjutnya, dilakukan *looping* yang akan berakhir ketika pembacaan sudah mencapai akhir dari *stream*. Hasil pembacaan akan ditampilkan di layar monior.

Setelah program pada contoh di atas dikompilasi dan dieksekusi, pada layar monitor akan muncul hasil sebagai berikut :

```
Baris yang dibaca = :Saya sedang belajar pemrograman
berorientasi obyek
Baris yang dibaca = :Pemrogramannya menggunakan Java
Baris yang dibaca = :Ternyata pemrograman dengan Java
sungguh mengasyikkan
```

8.2.2.2 Writer

Writer adalah *node stream* yang berfungsi melakukan penulisan data karakter ke tujuan data. Beberapa *method* Writer adalah sebagai berikut :

a. `public void write (char[] cbuf) throws IOException` : Menuliskan semua isi *array* karakter ke tujuan data.

b. `public abstract void write( char[] cbuf, int off, int len) throws IOException` : Menuliskan sebagian isi *array* karakter ke tujuan data, dengan parameter `off` adalah offset/nomor indeks di mana pembacaan karakter dimulai. Sedangkan *len* merepresentasikan panjang karakter yang akan dibaca.

c. `public void write (String str) throws IOException` : Menuliskan literal yang direferensi oleh obyek `str` ke tujuan data.

d. `public void write (String str, int off, int len` : Menuliskan literal yang direferensi oleh obyek `str` ke tujuan data, dimulai dari indeks ke-*off* sepanjang *len* karakter.

Penggunaan Writer ditunjukkan pada contoh program berikut ini.

```
import java.io.*;
public class WriteToFileExample{
   FileReader fr = null;
   BufferedReader bufReader = null;
   BufferedWriter bufWriter = null;
   FileWriter fw = null;

   public static void main(String[] args){
        WriteToFileExample wtfe = new WriteToFileExample();
   }

   public WriteToFileExample(){
      try{
         fr = new FileReader("source.txt");
         fw = new FileWriter("destination.txt");
         bufReader = new BufferedReader(fr);
         bufWriter = new BufferedWriter(fw);
         String hasil;
         hasil = bufReader.readLine();
         while(hasil!=null){
            System.out.println(hasil);
            bufWriter.write(hasil,0,hasil.length());
            bufWriter.flush();
            bufWriter.newLine();
            hasil = bufReader.readLine();
         }
         bufReader.close();
         bufWriter.close();
      }catch(IOException e){}
   }
}
```

Pada contoh di atas terdapat obyek yang membaca data dari file, yaitu obyek `fr` yang merupakan instans dari kelas `FileReader`. Selain itu,

juga terdapat obyek yang menulis data ke *file*, yaitu obyek fw yang merupakan instans dari kelas FileWriter. Atribut lainnya adalah obyek BufferedReader *bufReader* dan BufferedWriter bufWriter yang berfungsi untuk menampung data yang dibaca oleh fr dan menuliskan data tersebut ke fw.

Pada konstruktor WriteToFileExample() fr diinstanstiasi dengan menggunakan konstruktor FileReader(String src) di mana fr didefinisikan sebagai FileReader yang membaca data dari file **source.txt**. Obyek fw diinstanstiasi dengan menggunakan konstruktor FileWriter(String dest), di mana fw didefinisikan sebagai FileWriter yang menulis data ke file **destination.txt**. Selanjutnya, bufReader diinstanstiasi dengan menggunakan konstruktor BufferedReader(Reader in) di mana bufReader berfungsi untuk menampung data yang dibaca oleh fr. Atribut bufWriter diinstanstiasi dengan menggunakan konstruktor BufferedWriter(Writer out) di mana bufWriter berfungsi untuk menulis data ke fw.

Proses pembacaan file **source.txt** selanjutnya dilakukan dalam *looping*. Selama pembacaan bufReader belum mencapai akhir dari *stream*, maka proses pembacaan *file* diteruskan, dan hasilnya disalin ke variabel hasil. Pembacaan *file* tersebut dilakukan dengan memanggil *method* readLine() dari obyek bufReader.

Nilai variabel hasil akan ditulis ke file **destination.txt** dengan memanggil *method* write() dari obyek bufWriter. *Method* flush() dari obyek bufWriter dipanggil untuk memastikan bahwa penulisan data ke file tujuan sudah dilakukan. Setelah pembacaan dan penulisan data selesai dilakukan, dilakukan penutupan bufReader, dan bufWriter.

Untuk menjalankan contoh di atas, terlebih dahulu harus dipersiapkan *file* **source.txt** dan diisi dengan kalimat bebas, misalnya :

```
Saya sedang belajar Java
Ternyata pemrograman Java itu mudah
Percaya atau tidak ?
```

Keluaran dari program pada contoh di atas adalah sebuah *file* **destination.txt** yang berisi data :

```
Saya sedang belajar Java
Ternyata pemrograman Java itu mudah
Percaya atau tidak ?
```

9 NETWORKING

9.1 Konsep *Networking*

Konsep *networking* berkaitan dengan proses komunikasi data dari satu komputer ke komputer lain. Pertukaran data tersebut tidak hanya melibatkan jaringan satu komputer ke satu komputer (*peer-to-peer*), tetapi juga melibatkan jutaan komputer yang ada di dunia. Karena semakin banyaknya komputer yang terkoneksi ke internet, diperlukan aturan-aturan yang akan menjamin komunikasi data antar komputer.

Agar komunikasi data dapat dilakukan dengan baik, terdapat 2 hal yang penting yang harus didefinisikan dengan tepat, yaitu :

a. `IP address`, yang berkaitan dengan identifikasi unik sebuah komputer.
b. `port number`, yang berkaitan dengan jenis *service* yang digunakan.

9.1.1 *Addressing*

Sebuah komputer pada jaringan *internet* dapat diwakili oleh `IP address`-nya. Ketika sebuah paket data dikirimkan, paket tersebut memiliki *header* yang memuat informasi `IP address` tujuan. Melalui informasi ini, pada kondisi tanpa gangguan jaringan dan gangguan lainnya paket data tersebut dapat sampai ke komputer dengan `IP address` yang dimaksud.

9.1.2 *Port*

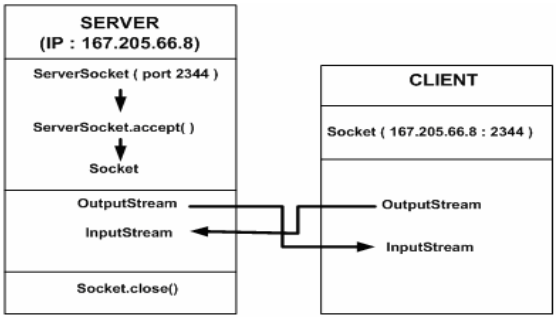
Nomor *Port* bernilai antara 0 - 65535. Paket data yang datang ke sebuah komputer akan diperiksa nomor *port*-nya, dan komputer akan menggunakan data tersebut untuk menjalankan *service*. *Service* yang berkaitan dengan jaringan *internet* akan menggunakan data yang paket datanya mencantumkan nomor *port* tertentu pada *headernya*. Misalnya, sebuah *service* A pada sebuah komputer akan menggunakan data yang dikirimkan dengan nomor *port* 7898. Maka setiap kali komputer tersebut menerima paket data yang nomor *port*-nya adalah 7898, data pada paket komputer akan digunakan oleh *service* A.

Salah satu *service* yang sudah umum digunakan dalam dunia internet adalah HTTP (*hypertext transport protocol*) yang menggunakan data dengan nomor *port* 8080.

9.2 Model Networking pada Teknologi Java

Teknologi Java mengakomodasi *networking* dengan menggunakan kelas `Socket` dan `ServerSocket`. Terminologi *socket* di sini menggambarkan pasangan (*address*, *port number*). *Socket* berfungsi sebagai "perwakilan" komputer untuk dapat mengadakan komunikasi dengan komputer lain. Dapat dikatakan bahwa komunikasi data yang dilakukan merupakan komunikasi antar-*socket*.

Untuk mengalirkan data dari satu komputer ke komputer lain, digunakan *Stream* (`InputStream` dan `OutputStream`). `Socket` dan `ServerSocket` memiliki `InputStream` dan `OutputStream` masing-masing sehingga masing-masing *socket* dapat saling berkirim data.



Gambar 9.1 Model *networking* pada Teknologi Java.

9.2.1 *Socket*

Socket merupakan representasi dari komputer yang menjadi tujuan data. *Socket* bukanlah *node stream*. Pada kelas `Socket` terdapat `InputStream` dan `OutputStream`. `InputStream` digunakan oleh *Socket* untuk mengambil data yang akan dikirimkan. Sedangkan `OutputStream` digunakan oleh *Socket* untuk mengirimkan data ke `ServerSocket`.

Di samping mempunyai `InputStream` dan `OutputStream`, *Socket* mendefinisikan `IP address` dan `port` komputer tujuan yang akan dikirimi data. Jadi, *Socket* lebih berperan dalam hal pengiriman data.

Socket digunakan pada komputer *client* untuk mengirimkan data atau *request* ke *server*. Seperti diperlihatkan pada Gambar 9.1, sebuah *client* akan menggunakan OutputStream dari *Socket* untuk mengirimkan data. OutputStream dari *Socket* pada *client* mengirimkan data ke InputStream milik *Socket* yang berada di *Server*.

Kelas *Socket* memiliki beberapa konstruktor, antara lain :

a. Socket() membuat sebuah *socket* yang belum dikoneksikan dengan *server*.
b. Socket(InetAddress inetAddress, int port) membuat sebuah *socket* dan mengkoneksikannya dengan *server* yang memiliki IP *Address* inetAddress dan momor *port* port.
c. Socket(String host, int port) membuat sebuah *socket* yang mengkoneksikannya dengan *server* dengan parameter nama host host dan nomor *port* port.

Untuk melakukan koneksi dengan ServerSocket, ada beberapa langkah yang dilakukan, yaitu :

a. **Melakukan instanstiasi obyek Socket** dengan menggunakan konstruktor-konstruktor yang telah dijelaskan sebelumnya.
b. **Melakukan instanstiasi obyek InputStream**. Obyek InputStream bukan obyek InputStream yang bebas diinstanstiasi sendiri, tetapi obyek InputStream yang merupakan milik dari obyek Socket. Oleh karena itu, instanstiasi dilakukan dengan memanggil *method* getInputStream() dari obyek Socket. Obyek InputStream tersebut digunakan untuk menerima data *response* dari *server*.
c. **Melakukan instanstiasi obyek OutputStream** yang diambil dari OutputStream milik Socket, dengan memanggil *method* getOutputStream(). Obyek OutputStream tersebut digunakan untuk mengirimkan data *request* ke *server*.
d. **Melakukan instanstiasi obyek InputStream atau Reader** yang membaca data dari sumber data pada komputer *client* (misalnya : *keyboard* / *file*).
e. **Melakukan pengiriman data dengan menggunakan obyek OutputStream** yang telah diinstanstiasi.
f. **Menerima *response* dari *server*,** menggunakan InputStream yang telah diinstanstiasi.

9.2.2 ServerSocket

ServerSocket adalah kelas yang khusus dibuat pada komputer yang akan menjalankan aplikasi *server*. ServerSocket meskipun berada pada *package* yang sama dengan Socket, posisinya dalam hierarki kelas tidak memiliki hubungan induk-anak dengan Socket. Hal ini disebabkan fungsi ServerSocket yang berbeda dengan Socket.

ServerSocket tidak digunakan untuk berkomunikasi data, tetapi hanya sebagai representasi *server* yang menunggu data yang akan masuk ke *server*.

Kelas ServerSocket mempunyai beberapa konstruktor, antara lain :

a. ServerSocket() membuat *server socket* yang belum ditetapkan nomor port-nya.
b. ServerSocket(int port) membuat *server socket* dengan nomor port yang sudah ditetapkan.

Untuk berkomunikasi 2 arah dengan *client*, sebuah *server* tetap membuat sebuah obyek Socket, tetapi obyek Socket tersebut diinstanstiasi dengan memanggil *method* accept() milik kelas ServerSocket.

Method accept() berfungsi untuk menunggu adanya koneksi dari *client*. Selama ServerSocket belum menerima sinyal koneksi dari *client*, program akan diam dan menunggu sampai menerima sinyal koneksi tersebut.

Untuk membentuk sebuah *server* yang dapat menerima *request* dan mengirimkan *response*, lakukan langkah-langkah berikut ini :

a. **Menginstanstiasi obyek ServerSocket** menggunakan konstruktor yang telah dijelaskan.
b. **Memanggil *method* accept() milik obyek ServerSocket** yang telah diinstanstiasi.
c. **Menginstanstiasi obyek Socket yang didapat ketika ServerSocket menerima *request* dari *client***. Obyek Socket ini menyimpan informasi tentang IP *address* dan nomor *port*-nya.
d. **Menangkap informasi yang dibawa oleh *request client*** dengan menggunakan kelas Reader/InputStream, misalnya BufferedReader.
e. **Melakukan proses pembentukan *response*.** Proses pembentukan *response* ini dapat berupa proses kalkulasi yang dilakukan di *server*. Hasil kalkulasi merupakan informasi *response*.
f. **Mengirimkan String/byte** ke *client* melalui OutputStream yang dimiliki oleh obyek Socket.
g. **Melakukan penutupan obyek Socket.**

9.3 Aplikasi *Client-Server* Sederhana

Aplikasi *Client-Server* Sederhana ini merupakan contoh komunikasi 2 arah antara *client* dan *server*. Spesifikasi aplikasi ini adalah sebagai berikut :

a. Aplikasi ini terdiri atas 2 kelas utama, yaitu kelas NetworkClient dan kelas NetworkServer.
b. Aplikasi NetworkClient berjalan di atas mesin yang berbeda dengan NetworkServer. Dalam contoh ini, NetworkClient berjalan di atas mesin dengan IP *address* 167.205.66.4. NetworkServer berjalan di atas mesin dengan IP *address* 167.205.66.5.
c. NetworkServer akan menunggu *request* dari *client* manapun.
d. NetworkClient akan melakukan koneksi ke NetworkServer lalu mengirimkan data ke *server* yang dituju.
e. *Server* akan memproses data yang dikirimkan padanya dan membuat *response*.
f. *Server* mengirimkan *respons*e ke *client*.

Kode program untuk *server* ditunjukkan pada contoh kode program berikut.

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

public class NetworkServer{
   private String inputLine;
   private String outputLine;
   private BufferedReader in;
   private ServerSocket serverSocket = null;
   private Socket clientSocket = null;
   private PrintWriter out = null;


   public static void main(String[] args){
       NetworkServer server = new NetworkServer();
   }

   public NetworkServer(){
      try{
         doAll();
         in.close();
      }catch(IOException e){}
   }

   public void doAll() throws IOException{
      serverSocket = new ServerSocket(4444);
      System.out.println("Server Socket " +
                         "Diinstanstiasi pada port " +
      serverSocket.getLocalPort());
      createConnection();
      readStream( );
   }

   public void createConnection() throws IOException{
      clientSocket = serverSocket.accept();
      System.out.println("Request Diterima dari : " +
      clientSocket.getInetAddress().getHostName());
      System.out.println("Port client : " +
      clientSocket.getPort());
```

```
            out = new PrintWriter(clientSocket.getOutputStream(),
                                  true);
            in = new BufferedReader(new InputStreamReader(
                                   clientSocket.getInputStream()));
        }

        public void readStream( ) throws IOException{
            while(true){
                try{
                    while((inputLine=in.readLine())!=null){
                        System.out.println(inputLine);
                        sendResponse(inputLine );
                    }
                }catch(IOException e){
                    createConnection();
                }
            }
        }

        public void sendResponse(String input) throws IOException{
            String words = "Anda mengirim pesan "+input;
            out.println(words);
            clientSocket.close();
        }
    }
```

Pada contoh program di atas konstruktor `NetworkServer()` memanggil method `doAll()`. Pada *method* `doAll()` dilakukan instanstiasi obyek `ServerSocket` menggunakan konstruktor `ServerSocket (int port)` dengan argumen `port` adalah 4444. hal ini berarti data yang diterima dari jaringan harus memiliki nomor *port* 4444 pada *header* paketnya agar dapat diproses oleh *server*. Obyek `ServerSocket` tersebut diberi nama `serverSocket`.

Setelah instanstiasi `serverSocket` selesai, langkah selanjutnya adalah menunggu *request* koneksi dari *client* dengan cara memanggil *method* `createConnection()`. *Method* ini memanggil *method* `accept()` milik obyek `serverSocket`. *Method* ini akan menunggu sampai ada *client* yang mengirimkan *request* koneksi. Ketika ada *request* koneksi yang datang, *method* `accept()` akan mengembalikan sebuah obyek `Socket` yang menangani informasi IP *address* dan nomor *port client*. Obyek `Socket` tersebut diberi nama `clientSocket`.

Langkah berikutnya pada *method* `createConnection()` adalah instanstiasi obyek `PrintWriter` bernama `out` yang menuliskan *response* ke *output stream* dari `clientSocket`. Setelah itu dilakukan instanstiasi obyek `BufferedReader` bernama `in` yang membaca data yang dikirimkan oleh *client*. Setelah *method* `createConnection()` selesai dieksekusi, *method* `doAll()` selanjutnya memanggil *method* `readStream()`.

Method readStream() mengandung skema *looping* while dengan nilai ekspresi boolean **true** sehingga *looping* akan terus dilakukan, kecuali jika terjadi eksepsi IOException, misalnya karena koneksi terputus. Di dalam *loop* while, terdapat blok try...catch yang berisi *loop* while dengan ekspresi boolean sebagai berikut.

```
(inputLine=in.readLine())!=null
```

hal ini berarti *loop* akan dijalankan apabila pembacaan data oleh obyek in tidak menghasilkan nilai null. Pembacaan data dilakukan dengan memanggil *method* readLine() milik obyek in. Data yang dibaca disimpan ke variabel inputLine untuk kemudian ditampilkan di layar monitor *server*.

Pada *method* readStream() dipanggil *method* sendResponse() dengan inputLine sebagai argumennya. *Method* sendResponse() akan membentuk *response* berupa String *words* yang akan menuliskan "Anda mengirim pesan ***** ", di mana ***** adalah nilai literal dari inputLine. Kemudian nilai *words* akan dituliskan ke *output stream* dari clientSocket.

Setelah data dikirimkan ke *client*, obyek clientSocket ditutup. Penutupan clientSocket ini menyebabkan data yang dituliskan ke *output stream* dari clientSocket akan dikirimkan ke *client*.

Penutupan clientSocket akan berakibat munculnya eksepsi IOException pada *method* readStream(). Tetapi, program di atas mengantisipasinya dengan memanggil kembali *method* createConnection() jika IOException muncul sehingga *server* menunggu adanya *request* lanjutan dari *client*. Hal ini disebabkan karena setelah *server* melakukan pengiriman *response clientSocket* otomatis ditutup, sehingga perlu dilakukan pembentukan koneksi kembali agar *server* dapat menerima *request* berikutnya dari *client*. Oleh karena itu, pada bagian catch *method* createConnection() dipanggil. Dengan demikian, *server* dapat terus bekerja sampai waktu tak terbatas. Kode program untuk *client* ditunjukkan berikut ini.

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

public class NetworkClient{
   private Socket socket = null;
   private PrintWriter out = null;
   private BufferedReader in = null;
   private BufferedReader stdIn = null;
   private String userInput;
   private BufferedReader responseReader = null;
```

```
    public static void main(String[] args){
       NetworkClient client = new NetworkClient();
    }
    public NetworkClient(){
       try{
          doAll();
       }catch(UnknownHostException e){
          e.getMessage();
          e.printStackTrace();
       }catch(IOException e2){
          e2.getMessage();
          e2.printStackTrace();
       }
    }

    private void createConnection() throws IOException,
                                    UnknownHostException{
       socket = new Socket("167.205.66.5",4444,
       InetAddress.getByName("167.205.66.4"),3333);
       System.out.println("Membangun Koneksi : ");
       System.out.println("Local Address : "+
       socket.getLocalAddress());
       System.out.println("Local Port Number : "+
       socket.getLocalPort());
       System.out.println("Remote Address "+
       socket.getInetAddress().getHostAddress());
       System.out.println("Remote Port Number "+
       socket.getPort());
       System.out.println("=======================\n");
       out = new PrintWriter(socket.getOutputStream(),true);
       in = new BufferedReader(new InputStreamReader(
                               socket.getInputStream()));
       stdIn = new BufferedReader(new
                               InputStreamReader(System.in));
    }

    public void doAll() throws IOException,
                                 UnknownHostException{
       createConnection();
       System.out.print("Masukkan Input / Request :  ");
       while((userInput=stdIn.readLine())!=null){
          out.println(userInput);
          receive( );
          createConnection();
       }
    }

    public void receive() throws IOException{
       String nextLine=in.readLine();
       System.out.println("\n\nMenerima Respons dari : ");
       System.out.println("Remote Address "+
       socket.getInetAddress().getHostAddress());
       System.out.println("Remote Port Number "+
       socket.getPort());
       System.out.print("Isi Respons : ");
       System.out.println(nextLine);
       System.out.println("====================\n");
       socket.close();
    }
}
```

Pada contoh program di atas konstruktor NetworkClient() memanggil *method* doAll(). *Method* doAll() memanggil *method* createConnection() untuk menjalin koneksi dengan *server*.

Method *createConnection*() menciptakan koneksi dengan menginstanstiasi obyek Socket dengan nama socket menggunakan konstruktor Socket(String remoteHostAddress, int remoteHostPort, String localHostAddress, int localHostPort). Konstruktor ini menyebabkan *client* mengirim *request* koneksi ke *server* dengan IP *address* remoteHostAddress dan nomor *port* remoteHostPort. Argumen localHostAddress adalah IP *address* dari *client*, dan localHostPort adalah *port* yang digunakan oleh *client* untuk mengirimkan data. Ketika melakukan spesifikasi nomor *port*, *client* menetapkan *port* localHostPort akan digunakan oleh *client* untuk mengirimkan data dan juga menjadi *port* tujuan bagi *server* untuk mengirimkan *response* ke *client*. Pada kasus di sini *client* akan mengirimkan *request* ke *server* dengan IP *address* 167.205.66.5 dan *port* 4444. Sedangkan *response* dari *server* akan diterima oleh *client* pada *port* 3333.

Method createConnection() menginstanstiasi obyek PrintWriter out yang akan menuliskan *request* ke *output stream* milik socket. Kemudian obyek BufferedReader in membaca *response* yang diterima dari *server*. Untuk menampung informasi yang berasal dari *keyboard*, *method* createControl() menginstanstiasi obyek BufferedReader stdIn yang membaca *stream* dari *keyboard*.

Setelah koneksi berhasil dibuat, *method* doAll() masuk ke skema *loop* while yang mempunyai ekspresi boolean sebagai berikut :

```
(userInput=stdIn.readLine())!=null
```

hal ini berarti *loop* akan terus dijalankan jika pembacaan dari *keyboard* belum mencapai akhir *stream*. Dalam *loop* while setiap *request* yang masuk dari *keyboard* akan dituliskan ke *output stream* milik socket. Setelah mengirimkan *request*, doAll() memanggil *method* receive() untuk menerima *response* dari *server*.

Method receive()membaca isi dari *input stream* dari *socket* dan menyalinnya ke variabel nextLine menggunakan *method* readLine(). *Method* readLine()memblok proses sampai *response* dari *server* masuk ke *client*. Setelah *response* diterima, *client* akan menampilkan *response* tersebut ke monitor. Setelah itu *socket* ditutup.

Setelah *method* receive() selesai dieksekusi, langkah selanjutnya adalah membentuk koneksi kembali dengan *server* untuk mengirimkan *request* berikutnya dengan memanggil *method* createConnection().

Output dari kedua aplikasi ini adalah sebagai berikut :

I. Inisialisasi pada *server* (tampilan di monitor *server*):

```
Server Socket Diinstanstiasi pada port 4444
```

II. *Client* membangun koneksi dengan *server* :

```
Membangun Koneksi :
Local Address : /167.205.66.4
Local Port Number : 3333
Remote Address 127.0.0.1
Remote Port Number 4444
====================================
Masukkan Input / Request :
```

III. Pengiriman *request* oleh *client* :

```
Masukkan Input / Request :  Ini Adalah Request [ENTER]
```

IV. Pengiriman *response* oleh *server* :

```
Request Diterima dari :
167.205.66.4
Port client : 3333
Anda mengirim pesan Ini Adalah Request
```

V. *Response* ditampilkan pada monitor *client* :

```
Menerima Respons dari :
Remote Address 167.205.66.5
Remote Port Number 4444
Isi Respons : Anda mengirim pesan Ini Adalah Request
====================================
```

DAFTAR PUSTAKA

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