BAB 9
Threads

9.1 Tujuan

Pada bab-bab sebelumnya Anda terbiasa untuk membuat program yang berururutan/sekuensial. Sebuah program sekuensial berarti sebuah program yang hanya memiliki satu aliran eksekusi. Setiap eksekusi, ia memiliki sebuah titik awal eksekusi, kemudian sebuah sekuen eksekusi, dan kemudian berakhir. Selama runtime, pasti hanya satu proses yang telah dieksekusi.

Bagaimanapun juga, di dunia nyata, pasti dibutuhkan sesuatu yang dapat mengatur proses yang terjadi dan berjalan bersama-sama.Oleh karena itu, thread hadir untuk menjadi solusi dalam mengatasi permasalahan tersebut.

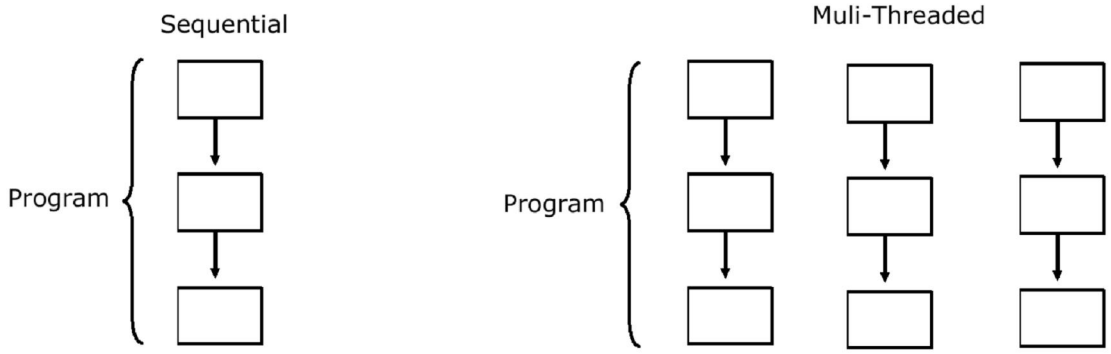
Pada akhir pembahasan, diharapkan pembaca dapat :

1. Mendefiniskan threads
2. Mengerti perbedaan state dalam threads
3. Mengerti konsep prioritas dalam threads
4. Mengetahui bagaimana menggunakan method didalam class Thread
5. Membuat sendiri sebuah thread
6. Menggunakan sinkronisasi pada thread yang bekerja bersama-sama dan saling bergantung satu dengan yang lainya
7. Memungkinkan thread untuk dapat berkomunikasi dengan thread lain yang sedang berjalan
8. Mengerti dan menggunakan kemampuan concurency

9.2 Definisi dan dasar-dasar thread

9.2.1 Definisi Thread

Sebuah thread merupakan sebuah pengontrol aliran program. Untuk lebih mudahnya, bayangkanlah thread sebagai sebuah proses yang akan dieksekusi didalam sebuah program tertentu. Penggunaan sistem operasi modern saat ini telah mendukung kemampuan untuk menjalankan beberapa program. Misalnya, pada saat Anda mengetik sebuah dokumen di komputer Anda dengan menggunakan text editor, dalam waktu yang bersamaan Anda juga dapat mendengarkan musik, dan surfing lewat internet di PC Anda. Sistem operasi yang telah terinstal dalam computer Anda itulah yang memperbolehkan Anda untuk menjalankan multitaskting. Seperti itu juga sebuah program (ibaratkan di PC Anda), ia juga dapat mengeksekusi beberapa proses secara bersama-sama(ibaratkan beberapa aplikasi berbeda yang bekerja pada PC Anda). Sebuah contoh aplikasi adalah HotJava browser yang memperbolehkan Anda untuk browsing terhadap suatu page, bersamaan dengan mendownload object yang lain, misalnya gambar, memainkan animasi, dan juga file audio pada saat yang bersamaan.



Gambar 1.1: Thread

9.2.2 State dari Thread

Sebuah thread memungkinkan untuk memiliki beberapa state:

1. Running
   Sebuah thread yang pada saat ini sedang dieksekusi dan didalam control dari CPU.
2. Ready to run
   Thread yang sudah siap untuk dieksekusi, tetapi masih belum ada kesempatan untuk melakukannya.
3. Resumed
   Setelah sebelumnya di block atau diberhentikan sementara, state ini kemudian siap untuk dijalankan.
4. Suspended
   Sebuah thread yang berhenti sementara, dan kemudian memperbolehkan CPU untuk menjalankan thread lain bekerja.
5. Blocked
   Sebuah thread yang di-block merupakan sebuah thread yang tidak mampu berjalan, karena ia akan menunggu sebuah resource tersedia atau sebuah event terjadi.

9.2.3 Prioritas

Untuk menentukan thread mana yang akan menerima control dari CPU dan akan dieksekusi pertama kali, setiap thread akan diberikan sebuah prioritas. Sebuah prioritas adalah sebuah nilai integer dari angka 1 sampai dengan 10, dimana semakin tinggi prioritas dari sebuah thread, berarti semakin besar kesempatan dari thread tersebut untuk dieksekusi terlebih dahulu.

Sebagai contoh, asumsikan bahwa ada dua buah thread yang berjalan bersama-sama. Thread pertama akan diberikan prioritas nomor 5, sedangkan thread yang kedua memiliki prioritas 10. Anggaplah bahwa thread pertama telah berjalan pada saat thread kedua dijalankan. Thread kedua akan menerima control dari CPU dan akan dieksekusi pada saat thread kedua tersebut memiliki prioritas yang lebih tinggi dibandingkan thread yang pada saat itu tengah berjalan. Salah satu contoh dari skenario ini adalah context switch.

Sebuah context switch terjadi apabila sebagian dari thread telah dikontrol oleh CPU dari

thread yang lain. Ada beberapa skenario mengenai bagaimana cara kerja dari context switch. Salah satu skenario adalah sebuah thread yang sedang berjalan memberikan kesempatan kepada CPU untuk mengontrol thread lain sehingga ia dapat berjalan. Dalam kasus ini, prioritas tertinggi dari thread adalah thread yang siap untuk menerima kontrol dari CPU. Cara yang lain dari context switch adalah pada saat sebuah thread yang sedang berjalan diambil alih oleh thread yang memiliki prioritas tertinggi seperti yang telah dicontohkan sebelumnya.

Hal ini juga mungkin dilakukan apabila lebih dari satu CPU tersedia, sehingga lebih dari satu prioritas thread yang siap untuk dijalankan. Untuk menentukan diantara dua thread yang memiliki prioritas sama untuk menerima kontrol dari CPU, sangat bergantung kepada sistem operasi yang digunakan. Windows 95/98/NT menggunakan time-slicing dan round-robin untuk menangani kasus ini. Setiap thread dengan prioritas yang sama akan diberikan sebuah jangka waktu tertentu untuk dieksekusi sebelum CPU mengontrol thread lain yang memiliki prioritas yang sama. Sedangkan Solaris, ia akan membiarkan sebuah thread untuk dieksekusi sampai ia menyelesaikan tugasnya atau sampai ia secara suka rela membiarkan CPU untuk mengontrol thread yang lain.

9.3 Class Thread

9.3.1 Constructor

Thread memiliki delapan constructor. Marilah kita lihat bersama beberapa constructor tersebut.

Constructor-constructor Thread
`Thread()`
Membuat sebuah object *Thread* yang baru.
`Thread(String name)`
Membuat sebuah object thread dengan memberikan penamaan yang spesifik.
`Thread(Runnable target)`
Membuat sebuah object *Thread* yang baru berdasar pada object *Runnable*. Target menyatakan sebuah object dimana method *run* dipanggil.
`Thread(Runnable target, String name)`
Membuat sebuah object *Thread* yang baru dengan nama yang spesifik dan berdasarkan pada object *Runnable*.

Tabel 1.2.1: Constructor dari Thread

9.3.2 Constants

Class *Thread* juga menyediakan beberapa constants sebagai nilai prioritas. Tabel berikut

ini adalah rangkuman dari class *Thread*.

Thread Constants
`public final static int MAX_PRIORITY`
Nilai prioritas maksimum, 10
`public final static int MIN_PRIORITY`
Nilai prioritas minimum, 1.
`public final static int NORM_PRIORITY`
Nilai default prioritas, 5.

Tabel 1.2.2:Konstanta dalam Thread

9.3.3 Methods

Method-method inilah yang disediakan dalam class *Thread*.

Method-method Thread
`public static Thread currentThread()`
Mengembalikan sebuah reference kepada thread yang sedang berjalan.
`public final String getName()`
Mengembalikan nama dari thread.
`public final void setName(String name)`
Mengulang pemberian nama thread sesuai dengan argument *name*. Hal ini dapat menyebabkan *SecurityException*.
`public final int getPriority()`
Mengembalikan nilai prioritas yang telah diberikan kepada thread tersebut.
`public final boolean isAlive()`
Menunjukkan bahwa thread tersebut sedang berjalan atau tidak.
`public final void join([long millis, [int nanos]])`
Sebuah overloading method. Sebuah thread yang sedang berjalan, harus menunggu sampai thread tersebut selesai (jika tidak ada parameter-parameter spesifik), atau sampai waktu yang telah ditentukan habis.
`public static void sleep(long millis)`
Menunda thread dalam jangka waktu milis. Hal ini dapat menyebabkan *InterruptedException*.
`public void run()`
Eksekusi thread dimulai dari method ini.
`public void start()`
Menyebabkan eksekusi dari thread berlangsung dengan cara memanggil method run.

Tabel 1.2.3: Method-method dari Thread

9.3.4 Sebuah contoh thread

Contoh dari thread pertama Anda adalah sebuah counter yang sederhana.

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

class CountDownGUI extends JFrame {
   JLabel label;
   CountDownGUI(String title) {
      super(title);
      label = new JLabel("Start count!");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      getContentPane().add(new Panel(), BorderLayout.WEST);
      getContentPane().add(label);
      setSize(300,300);
      setVisible(true);
   }
   void startCount() {
      try {
         for (int i = 10; i > 0; i--) {
            Thread.sleep(1000);
            label.setText(i + "");
         }
         Thread.sleep(1000);
         label.setText("Count down complete.");
         Thread.sleep(1000);
      } catch (InterruptedException ie) {
      }
      label.setText(Thread.currentThread().toString());
   }
   public static void main(String args[]) {
      CountDownGUI cdg = new CountDownGUI("Count down GUI");
      cdg.startCount();
   }
}
```

9.4 Membuat Threads

Sebuah thread dapat diciptakan dengan cara menurunkan (extend) class *Thread* atau dengan mengimplementasikan sebuah interface *Runnable*.

9.4.1 Menurunkan (extend) class Thread

Contoh berikut ini adalah user akan mendefinisikan sebuah class Thread yang akan menuliskan nama dari sebuah object thread sebanyak 100 kali.

```
class PrintNameThread extends Thread {
```

```
   PrintNameThread(String name) {
      super(name);
       // menjalankan thread dengan satu kali instantiate
      start();       }
   public void run() {
      String name = getName();
      for (int i = 0; i < 100; i++) {
         System.out.print(name);
      }
   }
}

class TestThread {
   public static void main(String args[]) {
      PrintNameThread pnt1 = new PrintNameThread("A");
      PrintNameThread pnt2 = new PrintNameThread("B");
      PrintNameThread pnt3 = new PrintNameThread("C");
      PrintNameThread pnt4 = new PrintNameThread("D");
   }
}
```

Perhatikan bahwa variable reference pnt1, pnt2, pnt3, dan pnt4 hanya digunakan satu kali. Untuk aplikasi ini, variabel yang menunjuk pada tiap thread pada dasarnya tidak dibutuhkan. Anda dapat mengganti body dari main tersebut dengan pernyataan berikut ini:

```
      new PrintNameThread("A");
      new PrintNameThread("B");
      new PrintNameThread("C");
      new PrintNameThread("D");
```

Program akan memberikan keluaran yang berbeda pada setiap eksekusi. Berikut ini adalah salah satu contoh dari output-nya.

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABCDABCDABCDA
BCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABC
DABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDBC
DBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBC
DBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBC
DBCDBCDBCDBCDBCDBCDBCD
```

9.4.2 Mengimplementasikan interface Runnable

Cara lain untuk membuat sendiri sebuah thread adalah dengan mengimplementasikan interface *Runnable*. Hanya satu method yang dibutuhkan oleh interface *Runnable* yaitu method run. Bayangkanlah bahwa method run adalah method utama dari thread yang Anda ciptakan.

Contoh dibawah ini hampir sama dengan contoh terakhir yang telah Anda pelajari, tapi pada contoh ini Anda akan mengimplement interface *Runnable*.

```
class PrintNameThread implements Runnable {
   Thread thread;
   PrintNameThread(String name) {
      thread = new Thread(this, name);
```

```
            thread.start();
        }
        public void run() {
            String name = thread.getName();
            for (int i = 0; i < 100; i++) {
                System.out.print(name);
            }
        }
    }

    class TestThread {
        public static void main(String args[]) {
            new PrintNameThread("A");
            new PrintNameThread("B");
            new PrintNameThread("C");
            new PrintNameThread("D");
        }
    }
```

9.4.3 Extend vs Implement

Dari dua cara untuk menciptakan thread seperti diatas, memilih salah satu dari kedua cara tersebut bukanlah sebuah permasalahan. Implement sebuah interface *Runnable* menyebabkan lebih banyak pekerjaan yang harus dilakukan karena kita harus mendeklarasikan sebuah object *Thread* dan memanggil method *Thread* dari object ini. Sedangkan menurunkan (extend) sebuah class Thread, bagaimanapun menyebabkan class Anda tidak dapat menjadi turunan dari class yang lainnya karena Java tidak memperbolehkan adanya multiple inheritance. Sebuah pilihan antara mudah tidaknya untuk diimplementasikan (implement) dan kemungkinan untuk membuat turunan (extend) adalah sesuatu yang harus Anda tentukan sendiri. Perhatikan mana yang lebih penting bagi Anda karena keputusan ada ditangan Anda.

9.4.4 Sebuah contoh penggunaan method join

Sekarang, pada saat Anda telah mempelajari bagaimana membuat sebuah thread, marilah kita lihat bagaimana method *join* bekerja. Contoh dibawah ini adalah salah satu contoh penggunaan method join tanpa argument. Seperti yang dapat Anda lihat, bahwa method tersebut (yang dipanggil tanpa argumen) akan menyebabkan thread yang sedang bekerja saat ini menungggu sampai thread yang memanggil method ini selesai dieksekusi.

```
class PrintNameThread implements Runnable {
   Thread thread;
   PrintNameThread(String name) {
      thread = new Thread(this, name);
      thread.start();
   }
   public void run() {
      String name = thread.getName();
      for (int i = 0; i < 100; i++) {
         System.out.print(name);
      }
   }
}

class TestThread {
   public static void main(String args[]) {
      PrintNameThread pnt1 = new PrintNameThread("A");
      PrintNameThread pnt2 = new PrintNameThread("B");
      PrintNameThread pnt3 = new PrintNameThread("C");
      PrintNameThread pnt4 = new PrintNameThread("D");
      System.out.println("Running threads...");
      try {
         pnt1.thread.join();
         pnt2.thread.join();
         pnt3.thread.join();
         pnt4.thread.join();
      } catch (InterruptedException ie) {
      }
      System.out.println("Threads killed."); //dicetak terakhir
   }
}
```

Cobalah untuk menjalankan program diatas. Apa yang Anda dapat? Melalui pemanggilan method join, kita memastikan bahwa pernyataan terakhir akan dieksekusi pada saat-saat terakhir.

Sekarang, berilah comment dilua blok try-catch dimana join dipanggil. Apakah ada perbedaan pada keluarannya?

9.5 Sinkronisasi

Sampai sejauh ini, Anda telah melihat contoh-contoh dari thread yang berjalan bersama-sama tetapi tidak bergantung satu dengan yang lainnya. Thread tersebut adalah thread yang berjalan sendiri tanpa memperhatikan status dan aktifitas dari thread lain yang sedang berjalan. Pada contoh tersebut, setiap thread tidak membutuhkan resource atau method dari luar sehingga ia tidak membutuhkan komunikasi dengan thread lain.

Didalam situasi-situasi tertentu, bagaimanapun sebuah thread yang berjalan bersama-sama kadang-kadang membutuhkan resource atau method dari luar. Oleh karena itu, mereka butuh untuk berkomunikasi satu dengan yang lain sehingga dapat mengetahui status dan aktifitas mereka. Contohnya adalah pada permasalahan produsen-konsumen. Kasus ini membutuhkan dua object utama, yaitu produsen dan konsumen. Kewajiban yang dimiliki oleh produsen adalah untuk membangkitkan nilai atau stream data yang diinginkan oleh konsumen.

9.5.1 Sebuah contoh yang tidak disinkronisasi

Marilah kita perhatikan sebuah kode sederhana yang mencetak sebuah string dengan urutan tertentu. Berikut ini adalah listing program tersebut :

```
class TwoStrings {
   static void print(String str1, String str2) {
      System.out.print(str1);
      try {
         Thread.sleep(500);
      } catch (InterruptedException ie) {
      }
      System.out.println(str2);
   }
}

class PrintStringsThread implements Runnable {
   Thread thread;
   String str1, str2;
   PrintStringsThread(String str1, String str2) {
      this.str1 = str1;
      this.str2 = str2;
      thread = new Thread(this);
      thread.start();
   }
   public void run() {
      TwoStrings.print(str1, str2);
   }
}
class TestThread {
   public static void main(String args[]) {
      new PrintStringsThread("Hello ", "there.");
      new PrintStringsThread("How are ", "you?");
      new PrintStringsThread("Thank you ", "very much!");
   }
}
```

Program ini diharapkan dapat mencetak dua argument object *Runnable* secara

berurutan. Permasalahannya adalah, pendeklarasian method *sleep* akan menyebabkan thread yang lain akan dieksekusi walaupun thread yang pertama belum selesai dijalankan pada saat eksekusi method print dari class *TwoStrings*. Berikut ini adalah contoh dari keluarannya.

```
Hello How are Thank you there.
you?
very much!
```

Pada saat berjalan, ketiga thread telah mencetak argument string pertama mereka sebelum argument kedua dicetak. Sehingga hasilnya adalah sebuah keluaran yang tidak jelas.

Sebenarnya, pada contoh diatas, tidak menunjukkan permasalahan yang serius. Akan tetapi pada aplikasi yang lain hal ini dapat menimbulkan exception atau permasalahan-permasalahan tertentu.

9.5.2 Mengunci Object

Untuk memastikan bahwa hanya satu thread yang mendapatkan hak akses kedalam method tertentu, Java memperbolehkan penguncian terhadap sebuah object termasuk method-method-nya dengan menggunakan monitor. Object tersebut akan menjalankan sebuah monitor implicit pada saat object dari method sinkronisasi dipanggil. Sekali object tersebut dimonitor, monitor tersebut akan memastikan bahwa tidak ada thread yang akan mengakses object yang sama. Sebagai konsekuensinya, hanya ada satu thread dalam satu waktu yang akan mengeksekusi method dari object tersebut.

Untuk sinkronisasi method, kata kunci yang dipakai adalah *synchronized* yang dapat menjadi header dari pendefinisian method. Pada kasus ini dimana Anda tidak dapat memodifikasi source code dari method, Anda dapat mensinkronisasi object dimana method tersebut menjadi anggota. Syntax untuk mensinkronisasi sebuah object adalah sebagai berikut:

```
synchronized (<object>) {
   //statements yang akan disinkronisasikan
}
```

Dengan ini, object dari method tersebut hanya dapat dipanggil oleh satu thread pada satu waktu.

9.5.3 Contoh Synchronized Pertama

Dibawah ini adalah kode yang telah dimodifikasi dimana method *print* dari class *TwoStrings* saat ini sudah disinkronisasi.

```
class TwoStrings {
   synchronized static void print(String str1, String str2) {
      System.out.print(str1);
      try {
         Thread.sleep(500);
      } catch (InterruptedException ie) {
      }
```

```
            System.out.println(str2);
        }
    }

    class PrintStringsThread implements Runnable {
        Thread thread;
        String str1, str2;
        PrintStringsThread(String str1, String str2) {
            this.str1 = str1;
            this.str2 = str2;
            thread = new Thread(this);
            thread.start();
        }
        public void run() {
            TwoStrings.print(str1, str2);
        }
    }

    class TestThread {
        public static void main(String args[]) {
            new PrintStringsThread("Hello ", "there.");
            new PrintStringsThread("How are ", "you?");
            new PrintStringsThread("Thank you ", "very much!");
        }
    }
```

Program tersebut saat ini memberikan keluaran yang benar.

```
Hello there.
How are you?
Thank you very much!
```

9.5.4 Contoh Synchronized Kedua

Dibawah ini adalah versi lain dari kode diatas. Sekali lagi, method *print* dari class *TwoStrings* telah disinkronisasi. Akan tetapi selain synchronized keyword diimplementasikan pada method,ia juga diaplikasikan pada object-nya.

```
class TwoStrings {
    static void print(String str1, String str2) {
        System.out.print(str1);
        try {
            Thread.sleep(500);
        } catch (InterruptedException ie) {
        }
        System.out.println(str2);
    }
}

class PrintStringsThread implements Runnable {
    Thread thread;
    String str1, str2;
    TwoStrings ts;
    PrintStringsThread(String str1, String str2, TwoStrings ts)
    {
```

```
            this.str1 = str1;
            this.str2 = str2;
            this.ts = ts;
            thread = new Thread(this);
            thread.start();
        }
        public void run() {
            synchronized (ts) {
                ts.print(str1, str2);
            }
        }
    }

    class TestThread {
        public static void main(String args[]) {
            TwoStrings ts = new TwoStrings();
            new PrintStringsThread("Hello ", "there.", ts);
            new PrintStringsThread("How are ", "you?", ts);
            new PrintStringsThread("Thank you ", "very much!", ts);
        }
    }
```

Program ini juga memiliki keluaran pernyataan-pernyataan yang benar.

9.6 Komunikasi antar thread (Interthread)

Pada bagian ini, Anda akan mempelajari mengenai method-method dasar yang digunakan thread untuk berkomunikasi dengan thread lain yang sedang berjalan.

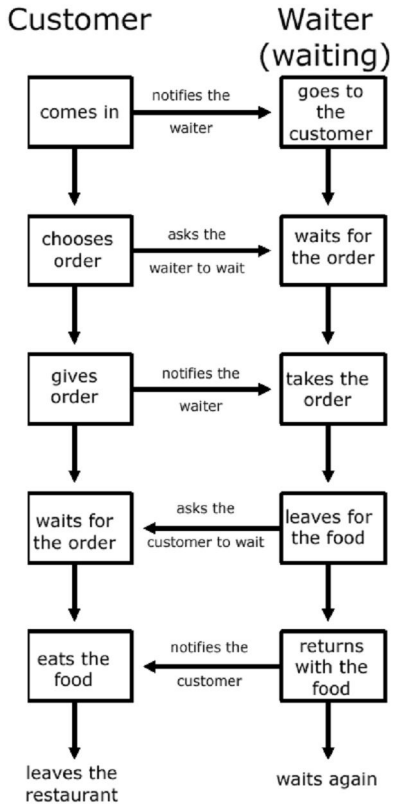
Method-method untuk komunikasi Interthread
`public final void wait()`
Menyebabkan thread ini menunggu sampai thread yang lain memanggil *notify* atau *notifyAll* method dari object ini. Hal ini dapat menyebabkan *InterruptedException*.
`public final void notify()`
Membangunkan thread yang telah memanggil method *wait* dari object yang sama.
`public final void notifyAll()`
Membangunkan semua thread yang telah memanggil method *wait* dari object yang sama.

Tabel 1.5: Methods untuk komunikasi Interthread

Untuk mendapatkan penjelasan dari method ini, perhatikanlah skenario pelayan-pelanggan. Pada skenario di sebuah restoran, seorang pelayan tidak akan menanyakan ke setiap orang apakah mereka akan memesan atau membutuhkan sesuatu, akan tetapi ia akan menunggu sampai pelanggan datang ke restoran tersebut. Pada saat seseorang datang, hal ini mengindikasikan bahwa ia mempunyai keinginan untuk memesan makanan dari restaurant tersebut. Atau juga dapat kita nyatakan bahwa pelanggan yang memasuki restaurant mengindikasikan (*notify*) bahwa pelayan dibutuhkan untuk memberikan pelayanan. Akan tetapi,dalam kondisi sepert ini, seorang pelanggan belum

siap untuk memesan. Akan sangat mengganggu apabila pelayan terus-menerus bertanya kepada pelanggan apakah ia telah siap untuk memesan atau tidak. Oleh karena itu, pelayan akan menunggu (*wait*) sampai pelanggan memberikan tanda (*notifies*) bahwa ia telah siap untuk memesan. Sekali pelanggan sudah memesan, akan sangat mengganggu apabila ia terus menerus bertanya kepada pelayan, apakah pesanannya sudah tersedia atau tidak. Normalnya, pelanggan akan menunggu sampai pelayan memberikan tanda (*notifies*) dan kemudian menyajikan makanan.

Perhatikan pada skenario berikut, setiap anggota yang menunggu, hanya akan berjalan sampai anggota yang lain memberi tanda yang memerintahkan untuk berjalan. Hal ini sama dengan yang terjadi pada thread.



Gambar 1.5: Skenario Pelayan-Pelanggan

9.6.1 Contoh Produsen-Konsumen

Contoh dibawah ini adalah salah satu implementasi dari permasalahan produsen-konsumen. Sebuah kelas yang menyediakan method untuk membangkitkan dan mengurangi nilai dari integer yang dipisahkan dari class Produsen dan Konsumen thread.

```
class SharedData {
   int data;
   synchronized void set(int value) {
      System.out.println("Generate " + value);
      data = value;
   }
   synchronized int get() {
      System.out.println("Get " + data);
      return data;
   }
}

class Producer implements Runnable {
   SharedData sd;
   Producer(SharedData sd) {
      this.sd = sd;
      new Thread(this, "Producer").start();
   }
   public void run() {
      for (int i = 0; i < 10; i++) {
         sd.set((int)(Math.random()*100));
      }
   }
}

class Consumer implements Runnable {
   SharedData sd;
   Consumer(SharedData sd) {
      this.sd = sd;
      new Thread(this, "Consumer").start();
   }
   public void run() {
      for (int i = 0; i < 10 ; i++) {
         sd.get();
      }
   }
}

class TestProducerConsumer {
   public static void main(String args[]) throws Exception {
      SharedData sd = new SharedData();
      new Producer(sd);
      new Consumer(sd);
   }
}
```

Dibawah ini adalah contoh dari keluaran program :

```
Generate 8
Generate 45
Generate 52
Generate 65
Get 65
Generate 23
Get 23
Generate 49
```

```
Get 49
Generate 35
Get 35
Generate 39
Get 39
Generate 85
Get 85
Get 85
Get 85
Generate 35
Get 35
Get 35
```

Hasil tersebut bukanlah hasil yang kita harapkan. Kita berharap bahwa setiap nilai yang diproduksi oleh produser dan juga kita akan mengansumsikan bahwa konsumen akan mendapatkan nilai tersebut. Dibawah ini adalah keluaran yang kita harapkan.

```
Generate 76
Get 76
Generate 25
Get 25
Generate 34
Get 34
Generate 84
Get 84
Generate 48
Get 48
Generate 29
Get 29
Generate 26
Get 26
Generate 86
Get 86
Generate 65
Get 65
Generate 38
Get 38
Generate 46
Get 46
```

Untuk memperbaiki kode diatas, kita akan menggunakan method untuk komunikasi interthread. Implementasi dibawah ini adalah implementasi dari permasalahan produsen konsumen dengan menggunakan method untuk komunikasi interthread.

```
class SharedData {
   int data;
   boolean valueSet = false;
   synchronized void set(int value) {
      if (valueSet) {  //baru saja membangkitkan sebuah nilai
         try {
            wait();
         } catch (InterruptedException ie) {
         }
      }
      System.out.println("Generate " + value);
      data = value;
```

```
            valueSet = true;
            notify();
        }
        synchronized int get() {
            if (!valueSet) {  //produsen belum men-set sebuah nilai
                try {
                    wait();
                } catch (InterruptedException ie) {
                }
            }
            System.out.println("Get " + data);
            valueSet = false;
            notify();
            return data;
        }
    }

    /* Bagian kode ini tidak ada yang berubah*/
    class Producer implements Runnable {
        SharedData sd;
        Producer(SharedData sd) {
            this.sd = sd;
            new Thread(this, "Producer").start();
        }
        public void run() {
            for (int i = 0; i < 10; i++) {
                sd.set((int)(Math.random()*100));
            }
        }
    }

    class Consumer implements Runnable {
        SharedData sd;
        Consumer(SharedData sd) {
            this.sd = sd;
            new Thread(this, "Consumer").start();
        }
        public void run() {
            for (int i = 0; i < 10 ; i++) {
                sd.get();
            }
        }
    }

    class TestProducerConsumer {
        public static void main(String args[]) throws Exception {
            SharedData sd = new SharedData();
            new Producer(sd);
            new Consumer(sd);
        }
    }
```

9.7 Kemampuan Concurrency

Dengan dirilisnya J2SE 5.0, telah tersedia kontrol threading yang baru dan juga tambahan fitur yang disebut concurrency. Fitur baru ini dapat ditemukan didalam package java.util.concurrent. Didalam sub bab ini, ada dua jenis fitur concurrency yang akan dijelaskan.

9.7.1 Interface Executor

Salah satu penambahan fitur mutakhir yang telah dibangun dalam aplikasi multithread adalah framework *Executor*. Interface ini termasuk didalam package java.util.concurrent, dimana object dari tipe ini akan mengeksekusi tugas-tugas dari *Runnable*.

Tanpa penggunaan interface ini, kita akan mengeksekusi tugas dari *Runnable* dengan cara menciptakan instance dari *Thread* dan memanggil method *start* dari object *Thread*. Kode dibawah ini mendemonstrasikan hal tersebut:

```
new Thread(<aRunnableObject>).start();
```

Dengan kemampuan dari interface yang baru ini, object *Runnable* yang telah diberikan akan dieksekusi menggunakan kode berikut ini:

```
<anExecutorObject>.execute(<aRunnableObject>);
```

Framework Executor ini berguna untuk aplikasi multithread, karena thread membutuhkan pengaturan dan penumpukan di suatu tempat, sehingga thread bisa saja sangat mahal. Sebagai hasilnya, pembangunan beberapa thread dapat mengakibatkan error pada memori. Salah satu solusi untuk mengatasi hal tersebut adalah dengan pooling thread. Didalam sebuah pooling thread, sebuah thread tidak lagi berhenti sementara akan tetapi ia akan berada dalam antrian didalam sebuah pool, setelah ia selesai melaksanakan tugasnya. Bagaimanapun, mengimplementasikan sebuah skema thread pooling dengan desain yang baik, tidaklah mudah dilakukan. Permasalahan yang lain adalah kesulitan untuk membatalkan atau mematikan sebuah thread.

Framework Executor merupakan salah satu solusi dari permasalahan ini dengan cara mechanic decoupling task submission mengenai bagaimana setiap tugas dijalankan, termasuk detail dari penggunaan thread, penjadwalan, dan sebagainya. Lebih disarankan untuk membuat thread secara eksplisit daripada membuat thread dan menjalankannya lewat method start yang telah diset untuk setiap task. Oleh karena itu lebih disarankan untuk menggunakan potongan kode berikut ini:

```
Executor <executorName> = <anExecutorObject>;
<executorName>.execute(new <RunnableTask1>());
<executorName>.execute(new <RunnableTask2>());
...
```

Dikarenakan *Executor* adalah sebuah interface, ia tidak dapat di-instantiate. Untuk menciptakan sebuah object *Executor*, ia harus membuat sebuah class yang mengimplementasikan interface ini atau dengan menggunakan factory method yang telah disediakan class *Executor*. Class ini juga tersedia didalam package yang sama seperti *Executor* interface. Class Executors juga menyediakan factory method untuk me-manage thread pool sederhana. Berikut ini adalah rangkuman dari beberapa factory methods:

Factory Method dari class Executor
`public static ExecutorService newCachedThreadPool()`
Menciptakan sebuah pool thread yang akan menciptakan thread sesuai yang dibutuhkan, atau ia akan menggunakan kembali thread yang telah dibangun sebelumnya, apabila tersedia. Sebuah method overloading, juga akan menggunakan object *ThreadFactory* sebagai argument.
`public static ExecutorService newFixedThreadPool(int nThreads)`
Menciptakan sebuah pool thread yang dapat digunakan kembali untuk membetulkan sebuah thread yang berada didalam antrian yang tidak teratur. Sebuah overloading method, akan menggunakan object *ThreadFactory* sebagai tambahan parameter.
`public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`
Menciptakan sebuah pool thread yang akan menjadwalkan command yang akan berjalan setelah diberikan sebuah delay, atau untuk mengeksekusi secara periodic. Sebuah overloading method, akan menggunakan object *ThreadFactory* sebagai tambahan parameter.
`public static ExecutorService newSingleThreadExecutor()`
Menciptakan sebuah Executor yang digunakan sebagai satu-satu-nya pelaksana dari sebuah antrian thread yang tidak teratur. Sebuah overloading method, juga akan menggunakan object *ThreadFactory* sebagai tambahan parameter.
`public static ScheduledExecutorService newSingleThreadScheduledExecutor()`
Menciptakan sebuah Executor thread yang akan menjadwalkan command untuk dijalankan setelah delay tertentu, atau dieksekusi secara periodic. Sebuah overloading method, juga akan menggunakan object *ThreadFactory* sebagai tambahan parameter

Tabel 1.1: Factory Method didalam class Executor

Pada saat sebuah tugas dari *Runnable* telah dieksekusi dan diselesaikan dengan control sebuah interface *Executor*. Untuk memberhentikan thread ini, kita dapat dengan mudah memanggil method shutdown dari interface tersebut seperti berikut ini:

```
executor.shutdown();
```

9.7.2 Interface Callable

Ingatlah kembali, bahwa ada dua cara untuk menciptakan sebuah thread. Kita dapat meng-extend sebuah class Thread atau meng-implement sebuah interface *Runnable*. Untuk menentukan teknik mana yang akan digunakan, kita akan melihat secara spesifik fungsi dari masing-masing teknik dengan cara meng-override method run. Penulisan method tersebut ditunjukkan seperti berikut ini:

```
public void run()
```

Kelemahan-kelemahan dari menciptakan thread dengan cara tersebut adalah:

1. Method run tidak dapat melakukan pengembalian hasil selama ia memiliki *void* sebagai nilai kembaliannya.
2. Method run mewajibkan Anda untuk mengecek setiap exception karena overriding method tidak dapat menggunakan klausa *throws*.

Interface *Callable* pada dasarnya adalah sama dengan interface Runnable tanpa kelemahan-kelemahan yang telah disebutkan diatas. Untuk mendapatkan hasil dari sebuah pekerjaan yang telah diselesaikan oleh Runnable, kita harus melakukan suatu teknik untuk mendapatkan hasilnya. Teknik yang paling umum adalah dengan membuat

sebuah instance variable untuk menyimpan hasilnya. Kode berikut ini akan menunjukkan bagaimana hal tersebut dilakukan.

```
public MyRunnable implements Runnable {
  private int result = 0;

  public void run() {
    ...
    result = someValue;
  }
      /* Hasil dari attribute ini dijaga dari segala sesuatu
      perubahan yang dilakukan oleh kode-kode lain yang
      mengakses class ini */

  public int getResult() {
    return result;
  }
}
```

Tulislah interface Callable, kemudian dapatkanlah hasil sesederhana yang ditampilkan pada contoh dibawah ini.

```
import java.util.concurrent.*;

public class MyCallable implements Callable {
  public Integer call() throws java.io.IOException {
    ...
    return someValue;
  }
}
```

Method call memiliki penulisan seperti berikut ini:

```
V call throws Exception
```

V adalah sebuah tipe generic yang berarti nilai pengembalian dari pemanggilan method tersebut adalah tipe data reference apapun. Anda akan mempelajari tentang tipe data generic di bab selanjutnya.

Masih ada lagi fitur-fitur concurrency dalam J2SE 5.0. Lihatlah lagi didalam dokumentasi API untuk mendapatkan informasi lebih detail lagi mengenai fitur-fitur yang lain.

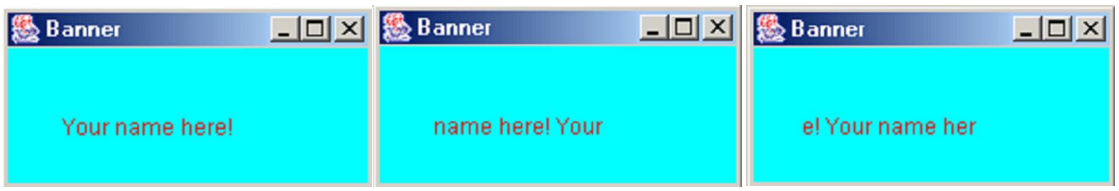
9.8 Latihan

9.8.1 Banner

Dengan menggunakan AWT atau Swing, buatlah sebuah banner sederhana yang akan

mencetak string yang dituliskan oleh user. String ini akan ditampilkan secara terus menerus dan program Anda harus memberikan ilustrasi bahwa string tersebut bergerak dari kiri ke kanan. Untuk memastikan bahwa proses perpindahannya tidak terlalu cepat, Anda sebaiknya menggunakan method sleep dari class Thread.

Berikut ini adalah sebuah contoh dimana Anda menuliskan ”Your name here!”.



Gambar 1.6.1: Contoh pergerakan string