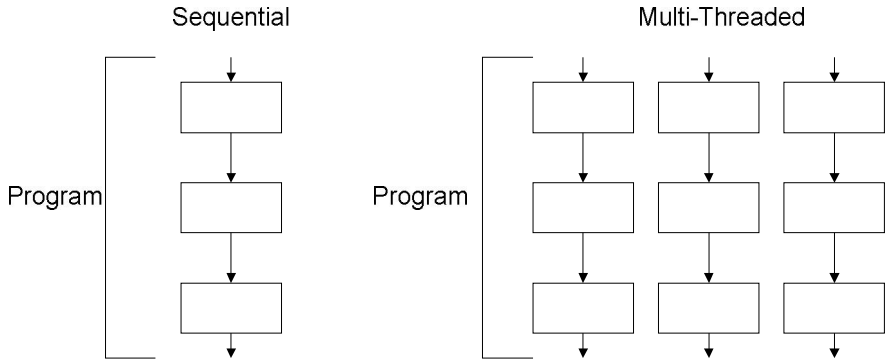
Pertemuan XIII
THREADS

13.1. Definisi dan Dasar-Dasar *Thread*
13.1.1. Definisi *Thread*

Thread merupakan kemampuan yang dimiliki oleh Java untuk membuat aplikasi yang tangguh, karena *thread* dalam program memiliki fungsi dan tugas tersendiri. Dengan adanya *thread*, dapat membuat program yang lebih efisien dalam hal kecepatan maupun penggunaan sumber daya, karena kita dapat membagi proses dalam aplikasi kita pada waktu yang sama. *Thread* umumnya digunakan untuk pemrograman *multitasking*, *networking*, dan lain-lain yang melibatkan pengaksesan ke sumber daya secara konkuren.

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 kita mengetik sebuah dokumen di komputer dengan menggunakan *text editor*, dalam waktu yang bersamaan kita juga dapat mendengarkan musik, dan *surfing* lewat internet di PC. Sistem operasi yang telah terinstal dalam komputer kita itulah yang memperbolehkan kita untuk menjalankan *multitaskting*. Seperti itu juga sebuah program (ibaratkan sebagai komputer), ia juga dapat mengeksekusi beberapa proses secara bersama-sama (ibaratkan beberapa aplikasi berbeda yang bekerja pada komputer). Sebuah contoh aplikasi adalah *HotJava browser* yang memperbolehkan kita untuk *browsing* terhadap suatu *page*, bersamaan dengan men-*download* obyek yang lain, misalnya gambar, memainkan animasi, dan juga file audio pada saat yang bersamaan.



Gambar 13.1. Perbandingan *Sequential* dan *Thread*

13.1.2. *State* dari *Thread*

Sebuah *thread* memungkinkan untuk memiliki beberapa *state* :

a. *Running*
   Sebuah *thread* yang pada saat ini sedang dieksekusi dan didalam kontrol dari CPU.
b. *Ready to run*
   Thread yang sudah siap untuk dieksekusi, tetapi masih belum ada kesempatan untuk melakukannya.

c. *Resumed*
   Setelah sebelumnya di blok atau diberhentikan sementara, *state* ini kemudian siap untuk dijalankan.
d. *Suspended*
   Sebuah *thread* yang berhenti sementara, dan kemudian memperbolehkan CPU untuk menjalankan *thread* lain bekerja.
e. *Blocked*
   Sebuah *thread* yang di blok merupakan sebuah *thread* yang tidak mampu berjalan, karena ia akan menunggu sebuah *resource* tersedia atau sebuah *event* terjadi.

13.1.3. Prioritas *Thread*

Untuk menentukan *thread* mana yang akan menerima kontrol 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 kontrol 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.

13.2. *Class Thread*

13.2.1. *Constructor*

Thread memiliki delapan *constructor*, berikut ini adalah beberapa constructor tersebut :

a. `Thread()`
   Membuat sebuah object *Thread* yang baru.

b. `Thread(String name)`
   Membuat sebuah object thread dengan memberikan penamaan yang spesifik.
c. `Thread(Runnable target)`
   Membuat sebuah object *Thread* yang baru berdasar pada object *Runnable*. Target menyatakan sebuah object dimana method *run* dipanggil.
d. `Thread(Runnable target, String name)`
   Membuat sebuah object *Thread* yang baru dengan nama yang spesifik dan berdasarkan pada object *Runnable*.
e. `Thread(ThreadGroup group, Runnable target)`
   Membuat sebuah object *Thread* yang baru dalam group.
f. `Thread(ThreadGroup group, Runnable target, String name)`
   Membuat sebuah object *Thread* yang baru dengan nama yang spesifik dan berdasarkan pada object *Runnable* dalam group.
g. `Thread(ThreadGroup group, Runnable target, String name, long stackSize)`
   Membuat sebuah object *Thread* yang baru dengan nama yang spesifik dan berdasarkan pada object *Runnable* dalam group dengan ukuran stack yang spesifik.
h. `Thread(ThreadGroup group, String name)`
   Membuat sebuah object thread dengan memberikan penamaan yang spesifik dalam group.

13.2.2. *Constants*

Class *Thread* juga menyediakan beberapa konstanta sebagai nilai prioritas. Berikut ini adalah rangkuman konstanta dari *class Thread*.

a. `public final static int MAX_PRIORITY`
   Nilai prioritas maksimum, 10
b. `public final static int MIN_PRIORITY`
   Nilai prioritas minimum, 1.
c. `public final static int NORM_PRIORITY`
   Nilai default prioritas, 5.

13.2.3. *Methods*

Metode-metode yang disediakan dalam *class Thread* antara lain :

a. `public static Thread currentThread()`
   Mengembalikan sebuah reference kepada thread yang sedang berjalan.
b. `public final String getName()`
   Mengembalikan nama dari thread.
c. `public final void setName(String name)`
   Mengulang pemberian nama thread sesuai dengan argument *name*. Hal ini dapat menyebabkan *SecurityException*.
d. `public final int getPriority()`
   Mengembalikan nilai integer yang menunjukkan prioritas yang telah diberikan kepada thread tersebut.
e. `public final boolean isAlive()`
   Mengembalikan nilai Boolean yang menunjukkan bahwa thread tersebut sedang berjalan atau tidak.

f. `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.
g. `public static void sleep(long millis)`
   Menunda thread dalam jangka waktu milis. Hal ini dapat menyebabkan *InterruptedException*.
h. `public void run()`
   Eksekusi thread dimulai dari method ini.
i. `public void start()`
   Menyebabkan eksekusi dari thread berlangsung dengan cara memanggil method run.

13.2.4. Sebuah Contoh *Thread*

Contoh dari *thread* pertama adalah sebuah counter yang sederhana.

```
import javax.swing.*;
import java.awt.*;
class CountDownGUI extends JFrame {
   JLabel label;
   JLabel label2;

   CountDownGUI(String title) {
      super(title);
      label = new JLabel("Start count!");
      label2 = 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();
   }
}
```

13.3. Membuat *Thread*

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

13.3.1. 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. Kita 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.

```
AAAAAAAAAAACCAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAACCCCCCBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDBBBBBBBBBBBBBBBBBBBBDDDDDDDDDDDDDD
```

13.3.2. 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 kita ciptakan.

Contoh di bawah ini hampir sama dengan contoh terakhir yang telah kita pelajari, tapi pada contoh ini kita akan mengimplementasikan *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 TestThreadRunnable {
  public static void main(String args[]) {
    new PrintNameThread("A");
    new PrintNameThread("B");
    new PrintNameThread("C");
    new PrintNameThread("D");
  }
}
```

13.3.3. *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 obyek ini.

Sedangkan menurunkan (*extend*) sebuah *class Thread*, bagaimanapun menyebabkan *class* yang kita buat 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 kita tentukan sendiri.

13.3.4. Sebuah contoh penggunaan *method join*

Sekarang, pada saat kita 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 kita 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 TestThreadJoin {
  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 kita dapat? Melalui pemanggilan *method join*, kita memastikan bahwa pernyataan terakhir akan dieksekusi pada saat-saat terakhir.

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

13.4. Sinkronisasi

Sampai sejauh ini, kita telah melihat contoh-contoh dari *thread* yang berjalan bersama-sama tetapi satu dengan yang lainnya tidak bergantung. *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 obyek utama, yaitu produsen dan konsumen. Kewajiban yang dimiliki oleh produsen adalah untuk membangkitkan nilai atau *stream* data yang diinginkan oleh konsumen.

13.4.1. 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 TestThreadNotSyn {
  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 obyek *Runnable* secara berurutan. Permasalahannya adalah, pendeklarasian metode sleep akan menyebabkan *thread* yang lain akan dieksekusi walaupun *thread* yang pertama belum selesai dijalankan pada saat eksekusi metode *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.

13.4.2. Mengunci Obyek

Untuk memastikan bahwa hanya satu *thread* yang mendapatkan hak akses kedalam metode tertentu, Java memperbolehkan penguncian terhadap sebuah obyek termasuk metode-metodenya dengan menggunakan monitor. Obyek tersebut akan menjalankan sebuah monitor implisit pada saat obyek dari metode sinkronisasi dipanggil. Sekali obyek tersebut dimonitor, monitor tersebut akan memastikan bahwa tidak ada *thread* yang akan mengakses obyek yang sama. Sebagai konsekuensinya, hanya ada satu *thread* dalam satu waktu yang akan mengeksekusi metode dari obyek tersebut.

Untuk sinkronisasi metode, kata kunci yang dipakai adalah *synchronized* yang dapat menjadi *header* dari pendefinisian metode. Pada kasus ini dimana kita tidak dapat memodifikasi *source code* dari metode, kita dapat mensinkronisasi obyek dimana metode

tersebut menjadi anggota. Sintaks untuk mensinkronisasi sebuah obyek adalah sebagai berikut:

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

Dengan ini, obyek dari metode tersebut hanya dapat dipanggil oleh satu *thread* pada satu waktu.

13.4.3. Contoh *Synchronized* Pertama

Dibawah ini adalah kode yang telah dimodifikasi dimana metode *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 TestThreadSyn1 {
  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!
```

13.4.4. Contoh *Synchronized* Kedua

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

```
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 TestThreadSyn2 {
  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);
  }
}
```

13.5. Komunikasi Antar *Thread* (*InterThread*)

Pada bagian ini, kita akan mempelajari mengenai metode-metode dasar yang digunakan *thread* untuk berkomunikasi dengan *thread* lain yang sedang berjalan.

Metode-metode untuk komunikasi *interthread* adalah :

a. `public final void wait()`
   Menyebabkan *thread* ini menunggu sampai *thread* yang lain memanggil *notify* atau *notifyAll* metode dari obyek ini. Hal ini dapat menyebabkan *InterruptedException*.

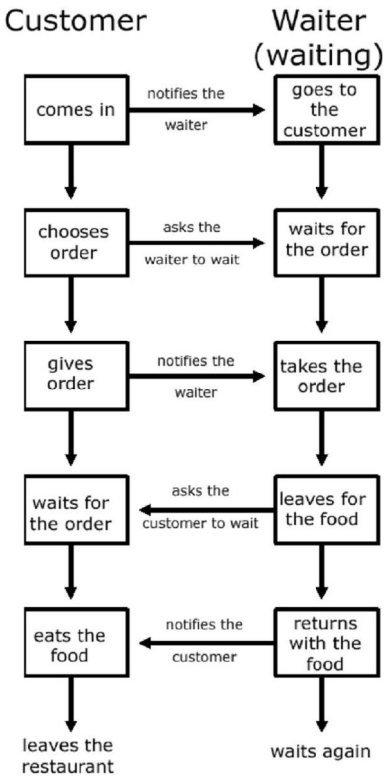
b. `public final void notify()`
   Membangunkan *thread* yang telah memanggil metode *wait* dari obyek yang sama.

c. `public final void notifyAll()`
   Membangunkan semua *thread* yang telah memanggil metode *wait* dari obyek yang sama.

Untuk mendapatkan penjelasan dari metode ini, perhatikanlah skenario pelayan-pelanggan pada gambar 9.2.

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 seperti 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 13.2. Skenario Pelayan-Pelanggan

13.5.1. Contoh Produsen-Konsumen

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

```
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);
  }
}
```

Di bawah ini adalah contoh keluaran dari program diatas.

```
Generate 1
Generate 88
Generate 37
Get 37
Get 37
Get 37
Get 37
Generate 23
Get 23
Get 23
Get 23
Get 23
Get 23
Get 23
Generate 88
Generate 97
Generate 11
Generate 31
Generate 53
Generate 81
```

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 metode untuk komunikasi *interthread*. Implementasi dibawah ini adalah implementasi dari permasalahan produsen konsumen dengan menggunakan metode 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 TestProducerConsumer2 {
  public static void main(String args[]) throws Exception {
    SharedData sd = new SharedData();
    new Producer(sd);
    new Consumer(sd);
  }
}
```

13.6. 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 di dalam *package java.util.concurrent*. Di dalam sub bab ini, ada dua jenis fitur *concurrency* yang akan dijelaskan.

13.6.1. *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 obyek 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 metode *start* dari obyek *Thread*.

Kode dibawah ini mendemonstrasikan hal tersebut:

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

Dengan kemampuan dari *interface* yang baru ini, obyek *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 detil dari penggunaan *thread*, penjadwalan, dan sebagainya. Lebih disarankan untuk membuat *thread* secara eksplisit daripada membuat *thread* dan menjalankannya lewat metode *start* yang telah di *set* 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 obyek *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 memanage *thread pool* sederhana.

Berikut ini adalah rangkuman dari beberapa *factory methods* :

a. `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 metode *overloading*, juga akan menggunakan obyek *ThreadFactory* sebagai *argument*.
b. `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 obyek *ThreadFactory* sebagai tambahan parameter.
c. `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 obyek *ThreadFactory* sebagai tambahan parameter.
d. `public static ExecutorService newSingleThreadExecutor()`
   Menciptakan sebuah *Executor* yang digunakan sebagai satu-satunya pelaksana dari sebuah antrian *thread* yang tidak teratur. Sebuah *overloading method*, juga akan menggunakan obyek *ThreadFactory* sebagai tambahan parameter.
e. `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.

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

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

13.6.2. *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 metode tersebut ditunjukkan seperti berikut ini :

```
public void run()
```

Kelemahan-kelemahan dari menciptakan *thread* dengan cara tersebut adalah:

a. *Method run* tidak dapat melakukan pengembalian hasil selama ia memiliki *void* sebagai nilai kembaliannya.

b. *Method run* mewajibkan kita 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 dibawah 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 metode tersebut adalah tipe data *reference* apapun.

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.

Referensi:

1. Hariyanto, Bambang, (2007), *Esensi-esensi Bahasa Pemrograman Java*, Edisi 2, Informatika Bandung, November 2007.
2. Utomo, Eko Priyo, (2009), *Panduan Mudah Mengenal Bahasa Java*, Yrama Widya, Juni 2009.
3. Tim Pengembang JENI, JENI 1-6, Depdiknas, 2007