JAVA CONCURRENCY

Berbicara mengenai *concurrency* artinya berbicara tentang *multi-thread*. Penerapan *multi-thread* sering anda jumpai meskipun tidak anda sadari, contohnya saja adalah aplikasi *word processor* (e.g.: Ms Word). Ketika anda bekerja dengan aplikasi *word processor*, aplikasi tersebut akan mencoba mendapatkan *inputan* dari pengguna dalam bentuk *text*, disaat yang sama pula aplikasi *word processor* akan memeriksa apakah *text* tersebut menggunakan bahasa (US style- biasanya) dengan benar. Selain itu, aplikasi juga akan melakukan proses *updating package* bila memang diperlukan. Menjalankan beberapa proses pada saat yang bersamaan disebut juga dengan *concurrency*.

Sebuah *thread* merupakan "a lightweight" proses yang ada di dalam sebuah aplikasi. Di dalam *thread* ini boleh memiliki sekumpulan proses, yang tidak dapat lagi dipecah menjadi unit-unit proses kecil lainnya, dan proses-proses ini saling bergantung satu dengan lainnya.

Penerapan *multi-thread* sendiri di Java *programming* dapat dilakukan dengan 2 cara, yaitu: melakukan meng-*inherit* dari kelas **Thread** atau mengimplementasi *interface* **Runnable**. Perlu anda ketahui bahwa kelas **Thread** sebenarnya mengimplementasikan *interface* **Runnable**. Jadi, secara tidak langsung anda telah melakukan implementasi terhadap *interface* tersebut.

Untuk menambah pemahaman anda mengenai penerapan *multi-thread* di Java *programming*, ketikan kode program berikut.

- Dengan cara meng-*inherit* kelas **Thread**

```
6   package simplethread;
7
8   import java.util.Random;
9
10  /**
11   *
12   * @author teamsar
13   */
14  public class SimpleThread extends Thread {
15      public SimpleThread(String nama){
16          super(nama);
17      }
18
19      public void run(){
20          for(int i=0; i<5; i++){
21              try{
22                  long time = getRandomNumberInRange(1, 10) * 1000;
23                  System.out.println(String.format("[%s] ini proses ke-%d membutuhkan waktu %d detik",
24                          this.getName(), i, time));
25                  Thread.sleep(time);
26              }catch(Exception e){
27                  System.out.println("Error " + e.getMessage());
28              }
29          }
30
31      }
32
33      /**
34       * @param args the command line arguments
35       */
36      public static void main(String[] args) {
37          // TODO code application logic here
38          new SimpleThread("Thread dari objek 1").start();
39          new SimpleThread("Thread dari objek 2").start();
40      }
41
42      private static int getRandomNumberInRange(int min, int max) {
43          if (min >= max) throw new IllegalArgumentException("max must be greater than min");
44          Random r = new Random();
45          return r.nextInt((max - min) + 1) + min;
46      }
47  }
48
```

Bila anda perhatikan kode program di atas, **proses** yang akan di-*handle* oleh si *thread* adalah "isi" di dalam *body* ***method*** *"run"*. Anda bebas menambahkan banyak proses di dalamnya (yang benar-benar *dependent* satu dengan lainnya, namun bila ada proses yang bisa di-*breakdown*, maka lebih baik dibuat *thread* lain untuk mempercepat proses komputasi), **dan itu hanya baru 1 thread saja.**

Thread lain yang "sengaja" dibentuk adalah dari objek yang ke-2. Hal ini dimaksudkan untuk menujukan konsep *multi-threading*.

Ketika anda membuat *thread* melalui kelas **Thread**, anda dapat memberi nama *thread* tersebut. Hal ini ditunjukan penggunaan *method* **super** di dalam konstruktor **SimpleThread.** Jadi, ketika anda mau mendapatkan nama *thread* ini selama ia berjalan, maka anda tinggal memanggil fungsi **getName** yang berada di dalam kelas **Thread.**

Berikut ini akan diberi contoh 2 kelas yang berbeda, ClassA dan ClassB, dimana masing-masing kelas menjalankan prosesnya (perbedaanya **hanya** pada banyaknya jumlah *looping*), namun akan dijalankan bersama-sama lewat *multi-threading*.

```
6   package multiprocessesdiffclasses;
7
8   import java.util.Random;
9
10  /**
11   *
12   * @author teamsar
13   */
14  public class ClassA extends Thread{
15      public ClassA(String nama){
16          super(nama);
17      }
18
19      public void run(){
20          System.out.println("Proses lain dari kelas A dijalankan..");
21          for(int i=0; i<5; i++){
22              try{
23                  long time = getRandomNumberInRange(1, 10) * 1000;
24                  System.out.println(String.format("[%s] ini proses ke-%d membutuhkan waktu %d detik",
25                          this.getName(), i, time));
26                  Thread.sleep(time);
27              }catch(Exception e){
28                  System.out.println("Error " + e.getMessage());
29              }
30          }
31      }
32
33      private static int getRandomNumberInRange(int min, int max) {
34          if (min >= max) throw new IllegalArgumentException("max must be greater than min");
35          Random r = new Random();
36          return r.nextInt((max - min) + 1) + min;
37      }
38  }
39
```

```java
package multiprocessesdiffclasses;

import java.util.Random;

/**
 *
 * @author teamsar
 */
public class ClassB extends Thread{
    public ClassB(String nama){
        super(nama);
    }

    public void run(){
        System.out.println("Proses lain dari kelas B dijalankan..");
        for(int i=0; i<10; i++){
            try{
                long time = getRandomNumberInRange(1, 10) * 1000;
                System.out.println(String.format("[%s] ini proses ke-%d membutuhkan waktu %d detik",
                        this.getName(), i, time));
                Thread.sleep(time);
            }catch(Exception e){
                System.out.println("Error " + e.getMessage());
            }
        }
    }

    private static int getRandomNumberInRange(int min, int max) {
        if (min >= max) throw new IllegalArgumentException("max must be greater than min");
        Random r = new Random();
        return r.nextInt((max - min) + 1) + min;
    }
}
```

```java
package multiprocessesdiffclasses;

/**
 *
 * @author teamsar
 */
public class MultiProcessesDiffClasses {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        ClassA objA = new ClassA("Objek A");
        ClassB objB = new ClassB("Objek B");

        objA.start(); // loop 0-4
        objB.start(); // loop 0-9
    }
}
```

- Dengan cara mengimplementasikan *interface* **Runnable**

```java
package threadbyrunnable;

import java.util.Random;

/**
 *
 * @author teamsar
 */
public class ThreadByRunnable implements Runnable{
    private String nama;

    public ThreadByRunnable(String nama){
        this.nama = nama;
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        new Thread(new ThreadByRunnable("Thread dari objek 1")).start();
        new Thread(new ThreadByRunnable("Thread dari objek 2")).start();
    }

    @Override
    public void run() {
        for(int i=0; i<5; i++){
            try{
                long time = getRandomNumberInRange(1, 10) * 1000;
                System.out.println(String.format("[%s] ini proses ke-%d membutuhkan waktu %d detik",
                        this.nama, i, time));
                Thread.sleep(time);
            }catch(Exception e){
                System.out.println("Error " + e.getMessage());
            }
        }
    }

    private static int getRandomNumberInRange(int min, int max) {
        if (min >= max) throw new IllegalArgumentException("max must be greater than min");
        Random r = new Random();
        return r.nextInt((max - min) + 1) + min;
    }
}

```

CHALLENGE!
Implementasikanlah *interface* **Runnable** untuk kasus 2 kelas, ClassA dan ClassB, seperti pada kode program di atas.

Ketika anda bekerja dengan beberapa *threads*, konkurensi menjamin pengerjaan *tasks* jauh lebih cepat dibandingkan dengan cara trandisional (tidak menggunakan *thread*). Secara teoritis, peningkatan terhadap kecepatan komputasi dapat dihitung dengan menggunakan aturan **Amdahl's Law**. Jika **ρ** merupakan persentase proses yang tidak dapat dijalankan secara *parallel* dan **N** merupakan banyaknya proses yang ada, maka peningkatan kecepatan dapat diformulasikan sebagai:

$\frac{\rho}{\left(\rho+\frac{1-\rho}{N}\right)}$ atau bila diberitahu waktu komputasi maka dapat dihitung dengan $\frac{1}{1-t}$, contoh:

bila sebuah program memerlukan waktu komputasi 20 jam dengan menggunakan *single processor core*, dan ada sebuah proses yang memakan waktu 1 jam **dan** tidak dapat diproses secara *parallel*, maka waktu yang tersisa yang bisa diparalelkan eksekusinya adalah 19 jam atau ***t = 0.95*** (*karena 1 proses memakan 0.05 persen dari keseluruhan proses*). Tanpa memerhatikan berapa banyak prosesor yang dipakai, eksekusi minimum dari aplikasi tersebut **tidak lebih kecil** dari 1 jam, sedangkan waktu yang bisa di-*speed up* komputasinya adalah sebesar: $\frac{1}{1-0.95}=20$ kali lipat.

PERMASALAHAN CONCURRENCY

Ada dua permasalahan mendasar yang ada di dalam *concurrency*, yaitu: *visibility problem* dan *accessibility problem*. Permasalahan *visibility* akan terjadi jika sebuah *thread* (misal A) membaca data yang di-*share*, dimana ada *thread* lain (misal B) mengubah-ubah isi data tersebut. *Thread* A tidak akan menyadari perubahaan tersebut. Sedangkan permasalahan *accessibility* akan terjadi bila beberapa *threads* mengakses data yang di-*share* secara bersamaan. Pengaruh kedua permasalahan konkurensi ini akan menyebabkan:

- *Liveness failure* – artinya adalah program tidak akan bereaksi lagi (menyebabkan *not responding*) akibat pengaksesan data secara bersamaan, atau disebut juga dengan **deadlock**.
- *Safety failure* – artinya adalah program akan menghasilkan *output* (data) yang salah, atau disebut juga dengan **dirty-read**.

Untuk mengatasi kedua permasalahan di atas, ada yang namanya mekanisme **sinkronisasi**. Di Java *programming*, mekanisme "locking" atau "mengkunci" untuk sebuah *method* atau *class* **sudah** disediakan, cukup dengan menggunakan kata kunci **synchronized**.

Kata kunci **synchronized** di Java *programming* memastikan:

- Hanya sebuah *thread* saja yang dapat mengeksekusi satu *block* kode program pada saat yang bersamaan.
- Masing-masing *thread* yang mengeksekusi satu *block* kode program tersebut, **akan** melihat perubahaan data dari aktifitas *thread-thread* sebelumnya.

Untuk menambah pemahaman anda mengenai konsep **sinkronisasi** ini, ketikan kode program di bawah ini:

```
6   package rabbitholeapp;
7
8   /**
9    *
10   * @author teamsar
11   */
12  public class Hole {
13      private int value;
14      private boolean isAvailable;
15
16      public synchronized int getValue(){
17          while(!isAvailable){
18              try{
19                  this.wait();
20              }catch(InterruptedException ie){
                    ie.printStackTrace();
22              }
23          }
24          this.isAvailable = false;
25          this.notifyAll();
26          return value;
27      }
28
29      public synchronized void putValue(int value){
30          while(isAvailable){
31              try {
32                  this.wait();
33              } catch (InterruptedException ie) {
                    ie.printStackTrace();
35              }
36          }
37          this.isAvailable = true;
38          this.value = value;
39          this.notifyAll();
40      }
41  }
```

```java
import java.util.Scanner;

/**
 *
 * @author teamsar
 */
public class Human implements Runnable{
    private volatile boolean exit = false;
    private Thread currThread = null;
    private Hole hole;

    public Human(Hole hole){
        this.hole = hole;
    }

    @Override
    public void run(){
        while(!exit){
            try{
                Scanner _scanner = new Scanner(System.in);
                System.out.println("Enter any number ranging from 0-9 to get your lovely rabbit!");

                int num = Integer.valueOf(_scanner.nextLine());

                if(num < 0) this.stop();
                else hole.putValue(num);

            }catch(Exception ex){
                System.out.println(ex.getMessage());
            }
        }
    }

    public void stop(){
        exit = true;
    }

    void start() {
        if(currThread == null){
            currThread = new Thread(this);
            currThread.start();
        }
    }
}
```

```java
import java.util.Random;

/**
 *
 * @author teamsar
 */
public class Rabbit extends Thread{
    private Hole hole;

    public Rabbit(Hole hole){
        this.hole = hole;
    }

    public void run(){
        while(true){
            try {
                int randNum = getRandomNumberInRange(0, 9);
                if(randNum == this.hole.getValue()) System.out.println("Got it! "
                        + "Your rabbit is in your bag now.");
                else System.err.println(String.format("Sorry, you missed the rabbit. "
                        + "The rabbit needs number %d. Please try again!", randNum));
            } catch (Exception ie) {
                ie.printStackTrace();
            }
        }
    }

    private static int getRandomNumberInRange(int min, int max) {
        if (min >= max) throw new IllegalArgumentException("max must be greater than min");
        Random r = new Random();
        return r.nextInt((max - min) + 1) + min;
    }
}
```

```java
package rabbitholeapp;

/**
 *
 * @author teamsar
 */
public class RabbitHoleApp {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        Hole objHole = new Hole();
        Human objHuman = new Human(objHole);
        Rabbit objRabbit = new Rabbit(objHole);

        objHuman.start();
        objRabbit.start();
    }

}

```

CHALLENGE!

- Modifikasikanlah program *RabbitHole* di atas, dengan **menambah kelas baru** yang bertugas untuk menghitung banyaknya *rabbit* yang berhasil anda tangkap, setiap kali anda berhasil menebak angka acak.
- Buatlah sebuah program sederhana untuk menunjukan teori **Amdahl's Law** benar atau tidak! (tips: buat dahulu program yang melakukan komputasi tanpa *multi-thread*, ambil waktu yang dibutuhkan. Kemudian implementasi dengan *multi-thread*, lalu ambil waktunya. Buat proses *dummy* yang tidak bisa diparalelkan prosesnya. Bagi waktu yang pertama dengan waktu yang sudah diterapkan *multi-thread*).
- Lakukan riset kecil, cari tahu apa itu *volatile* dan apa hubungannya dengan *mutual exclusive lock*.
- Lakukan observasi kecil untuk kelas **ExecutorService** (aka Thread pools in Java) dan juga **Semaphore** (cara lain melakukan sinkronisasi). Setelah itu buatlah aplikasi sederhana yang membaca data transaksi bank yang telah dilampirkan di cis.del.ac.id. Implementasikan konsep *threadpools* dan **Semaphore** (cara lain melakukan sinkronisasi terhadap *accessing data* yang di-*share)*. Note: program anda **harus** berbeda pengimplementasiannya dengan rekan yang lain. Anda bebas melakukan manipulasi data apapun di dalam program anda, misal: melakukan pengurangan **WITHDRAWAL AMT** dengan **BALANCE AMT** dari *bank account* tertentu.