Tutorial 7 Part 1

Praktikum Pemrograman Berbasis Objek

Asisten IF2210 2022/2023

Outline

1. Kenapa threading?
2. Penggunaan Threading
3. Wait dan Notify
4. Synchronized
5. Timer

Multithread

Multithread

Misalnya kalian lagi buat tugas besar 2 dengan JavaFX:

- Kalian perlu mengupdate waktu setiap detik untuk jam.
- Jika sleep nya kecil maka ada waktu dimana UI akan freeze. Dan update waktu jadi terlalu sering.
- Selain itu, saat membuat laporan, jika data banyak, dan pemrosesan pembuatan pdf lama, UI akan freeze saat memproses pembuatan laporan.
- Nah, untuk menghindari hal tersebut, kalian memerlukan thread lain untuk memproses hal tersebut. Agar main thread/thread khusus UI tidak terganggu dan UI dapat berjalan dengan normal meskipun sedang memproses hal lain.

Multithread

Itu hanya satu *use case* saja.

Multithread sangat banyak dimanfaatkan di programming, terutama saat satu task bottleneck oleh task lain, padahal keduanya dapat dijalankan bersamaan.

Ibaratnya, kalau kalian nge-carry tugas sendirian itu single thread. Saat kalian bagi tugas itu kalian multi thread. Tentu mengerjakan sendiri dengan bersama kelompok akan ada masalah-masalah baru. Seperti task yang saling berkaitan, saling tunggu-menunggu pekerjaan teman, proses tukar menukar informasi, kontrak kerja antar tugas, pembagian tugas dan sebagainya.

Multithread

Threading dapat dilakukan dengan 2 cara:

- Membuat kelas yang extends Thread
- Membuat kelas yang implements Runnable

Extends Thread

```
class FileDownloaderThread extend Thread {
  // Override
  public void run() {
    // Tuliskan fungsi yang akan dijalankan oleh thread
    // dalam method run
    downloadFile();
  }
}
```

```
...
Thread fileDownloader = new FileDownloaderThread();
// Untuk menjalankan thread
fileDownloader.start();
...
```

Pro-Tip: Program yang akan dijalankan dalam thread lain adalah program yang terdapat pada method **run()** yang dapat di-*override*.

Implements Runnable

```
class FileDonwloader implements Runnable {
  // Override
  public void run() {
    // Tuliskan fungsi yang akan dijalankan oleh thread
    // dalam method run
    downloadFile();
  }
}
```

```
...
Runnable fileDownloader = new FileDownloader();
Thread t1 = new Thread(fileDownloader);
// Untuk menjalankan thread
t1.start();
...
```

Extends vs Implements

Alasan menggunakan *implement* Runnable:

1. Java tidak memperbolehkan kita *extend* banyak kelas, jadi jika kita hanya bisa meng-*extend* kelas Thread saja.
2. Dengan *implement* Runnable kita hanya perlu menginstasikannya sekali untuk menjalankannya dalam banyak Thread

```
Runnable runThis = new SimpleRunnable();
Thread t1 = new Thread(runThis);
Thread t2 = new Thread(runThis);
t1.start();
t2.start();
```

Alasan menggunakan *extend* Thread:

1. *Extend* Thread memungkinkan kita untuk mengubah cara kerja Thread tersebut, seperti menambahkan prosedur baru sebelum Thread di **start()**

Wait and Notify

Wait and Notify

Bayangkan kalian diminta membuat sebuah game yang butuh 10 orang untuk dimulai.

Saat invoke start(), Game menunggu 10 orang join game.

Perhatikan kode berikut

```
public class Game {
  private int playerCount;

  public Game() {
      this.playerCount = 0;
  }

  public synchronized void onPlayerJoin() {
      int prevPlayerCount = this.playerCount;
      this.playerCount = prevPlayerCount + 1;
  }

  public synchronized void start() {
      for (int i = 0; i < 10; i++) {
        // Misalnya, proses untuk player join butuh waktu lama
          Thread.sleep(2000);
          this.onPlayerJoin();
      }
      System.out.println("starting game!");
  }
}
```

Wait and Notify

Tapi ini buruk! Karena misal untuk tiap player join butuh 2 detik, maka harus dibutuhkan 20 detik untuk game bisa dimulai!

Karena itu, kita mau memisahkan proses join player menjadi sebuah thread:

Wait and Notify

```
public class PlayerJoining extends Thread {
    private OnPlayerJoiningListener listener;

    public PlayerJoining(OnPlayerJoiningListener listener) {
        this.listener = listener;
    }

    public void run() {
        // Misalnya, proses untuk player join butuh waktu lama
        Thread.sleep(2000);
        listener.onPlayerJoin();
    }

    public interface OnPlayerJoiningListener {
        void onPlayerJoin();
    }
}
```

Wait and Notify

```
public class Game implements PlayerJoining.OnPlayerJoiningListener {
    private int playerCount;

    public Game() {
        this.playerCount = 0;
    }

    public void onPlayerJoin() {
        int prevPlayerCount = this.playerCount;
        this.playerCount = prevPlayerCount + 1;
    }

    public void start() {
        while (this.playerCount < 10) {
            // busy waiting, ini akan memakan proses sangat berat
        }
        System.out.println("starting game!");
    }
}
```

Wait and Notify

Kode tadi jelek, karena ada busy waiting. Resource processor akan termakan hanya untuk loop itu.

Masalah ini dapat kita selesaikan dengan ***wait*** dan ***notify***

Wait and Notify

wait()

Membuat thread yang memanggil fungsi ini menunggu.

notify()

Membuat semua thread yang melakukan ***wait()*** pada objek yang sama di-*resume*.

Pro-tips: ***wait()*** dan ***notify()*** beroperasi pada konteks objek. Jika t1 (Thread) melakukan ***wait()*** pada objek A dan t2 (thread) melakukan ***notify()*** pada objek B maka t1 tidak akan di-*resume*.

Wait and Notify

```
public class Game implements PlayerJoining.OnPlayerJoiningListener {
    private int playerCount;

    public Game() {
        this.playerCount = 0;
    }

    public void onPlayerJoin() {
        int prevPlayerCount = this.playerCount;
        this.playerCount = prevPlayerCount + 1;
        this.notify();
    }

    public void start() {
        while (this.playerCount < 10) {
            this.wait(); // kode ini blocking, artinya prosesor tidak akan
                         // melanjutkan kecuali ada yang melakukan notify
        }
        System.out.println("starting game!");
    }
}
```

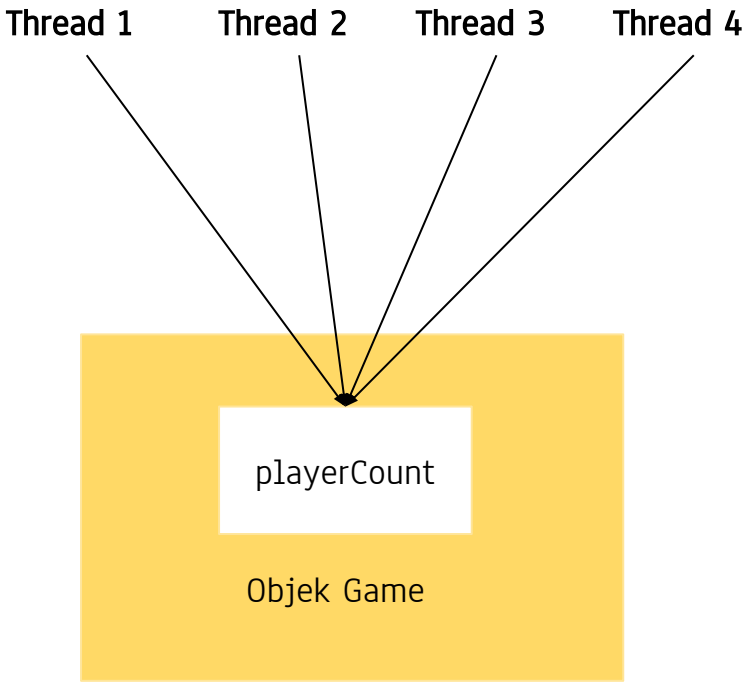
Wait and Notify

Coba jalankan kode tadi di laptop kalian. Dari awalnya 20 detik, sekarang hanya butuh 2 detik :)

Tapi, perhatikan ada kesalahan pada kode Game:

```
public void onPlayerJoin(String name) {
    int prevPlayerCount = this.playerCount;
    this.playerCount = prevPlayerCount + 1;
    this.notify();
}
```

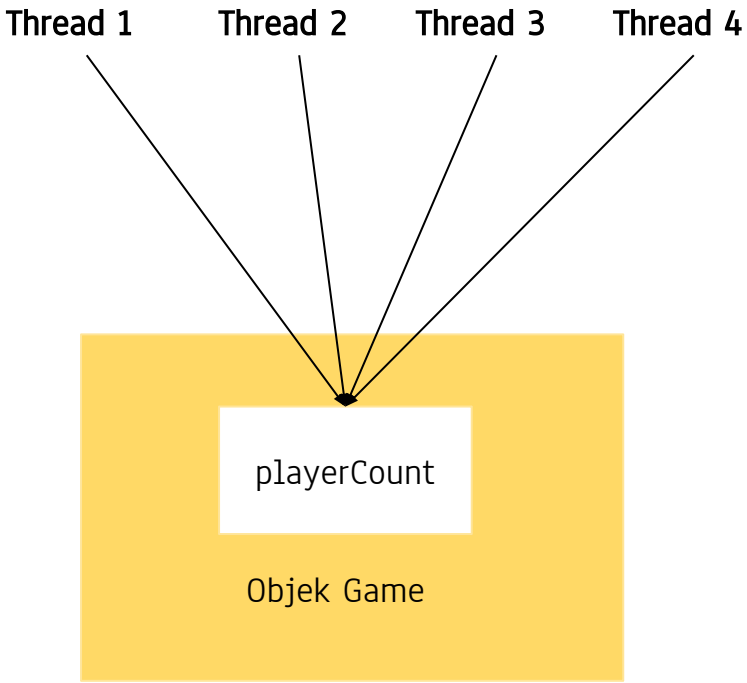
Sudah menemukan kesalahannya?



Thread 1-10 mengakses variabel *counter* yang sama pada objek Runner. Hal ini mereka lakukan ketika memanggil

```
listener.onPlayerJoin()
```

Pada fungsi *onPlayerJoin()* terdapat operasi *increment*. Apa yang terjadi jika operasi *increment* dilakukan dua thread yang berbeda secara bersamaan?



Contoh kasus:

Nilai playerCount = 5
Thread 1 membaca nilai playerCount // 5
Thread 2 membaca nilai playerCount // 5
Thread 1 menambah nilai playerCount // 6
Thread 2 menambah nilai playerCount // 6
Thread 1 mengassign nilai playerCount
Nilai playerCount = 6
Thread 2 mengassign nilai playerCount
Nilai playerCount = 6

Padahal nilai counter seharusnya menjadi 7 ketika dua buah Thread mengakses *onPlayerJoin()*

Synchronized

Synchronized

Mengubah kode menjadi

```
public void onPlayerJoin(String name) {
    this.playerCount++;
    this.notify();
}
```

akan menghasilkan masalah yang sama, karena pada dasarnya prosesor akan membreakdown operator increment (++) menjadi 3 step:

1. membaca nilai lama
2. membuat nilai baru yang merupakan nilai lama tambah 1
3. menyimpan nilai baru

Synchronized

Kasus ini disebut sebagai race condition, ketika 2 thread berebut resource dan menyebabkan perilaku yang tidak diharapkan.

Karena itu, kita bisa memanfaatkan keyword synchronized di java.

Synchronized memastikan agar java tidak akan menjalankan sebuah method dari lebih dari 1 thread sekaligus. Artinya, ketika sebuah method dipanggil oleh sebuah thread, thread lain harus menunggu sampai thread itu selesai menginvoke method itu.

Hasil revisinya ada di slide selanjutnya

Revisi Wait and Notify

```
public class Game implements PlayerJoining.OnPlayerJoiningListener {
    private int playerCount;

    public Game() {
        this.playerCount = 0;
    }

    public synchronized void onPlayerJoin(String name) {
        int prevPlayerCount = this.playerCount;
        this.playerCount = prevPlayerCount + 1;
        this.notify();
    }

    public synchronized void start() {
        while (this.playerCount < 10) {
            this.wait();
        }
        System.out.println("starting game!");
    }
}
```

Synchronized

Tips: Jangan meremehkan masalah multi-threading! Race condition, deadlock, dan berbagai macam masalah seperti ini masih sering dijumpai.

Terkadang, kesalahan ada di programmer yang tidak teliti. Misal, bagaimana jika ada 2 thread menambahkan data ke ArrayList secara bersamaan? Kita perlu memeriksa apakah library yang kita gunakan juga *thread safe* (artinya, aman jika diakses oleh beberapa thread sekaligus).

Timer

Timer

Timer adalah kelas util dari java yang digunakan untuk menjadwalkan aksi secara **periodik** yang akan dijalankan di thread.

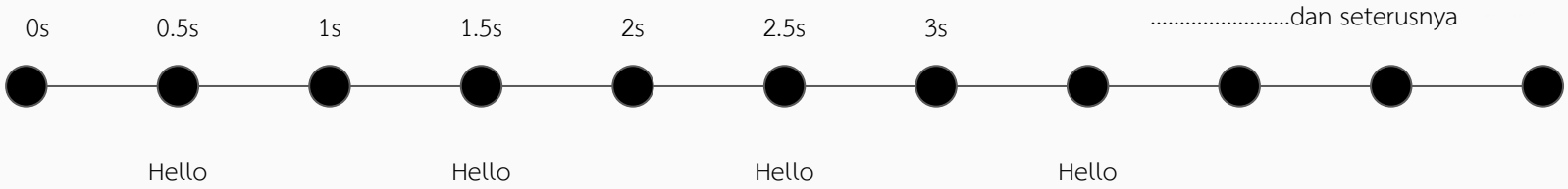
Aksi yang dijadwalkan ini ada didalam kelas **TimerTask**. Dalam kelas ini ada method **run()** yang merupakan aksi yang dijalankan.

Kode berikut akan mencetak “Hello” setiap 1000 ms (1 detik) dan akan jalan dimulai setelah 500ms (0.5 detik)

```
import java.util.Scanner;
import java.util.Timer;
import java.util.TimerTask;

public class Main {
  public static void main(String[] args) {
    Timer timer = new Timer();
    TimerTask timerTask = new TimerTask() {
      public void run() {
        System.out.println("Hello");
      }
    };
    timer.schedule(timerTask, 500, 1000);
  }
}
```

Timer: Contoh



Sekian.

Ditunggu praktikum dan tutorial berikutnya.