MINGGU 12

Praktikum Pemrograman Berbasis Objek
oleh Asisten IF2210 2019/2020

OUTLINE

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

MULTITHREAD

- Kenapa threading?
  - Kasus 1
    - Misal Anda membuat kalkulator dengan javafx
    - Lalu button = membutuhkan 2 detik untuk menghitung
    - Artinya, selama 2 detik tersebut, kalkulator akan *hang* (tidak ada button yang bisa diklik, dan lainnya) karena prosesor sedang fokus menghitung
    - Untuk itu, javafx memisahkan thread UI dan program

MULTITHREAD

- Kenapa threading?
  - Kasus 2
    - Misal Anda membuat sebuah backend sebuah website.
    - Jika tidak multithread, maka ketika ada 1 orang mengakses website, orang lain harus menunggu thread untuk orang itu selesai sebelum orang selanjutnya dilayani.
  - Dan banyak lagi
  - Multithread sangat banyak dimanfaatkan di programming, terutama saat satu task bottleneck oleh task lain, padahal keduanya dapat dijalankan bersamaan.

MULTITHREAD

- Untuk melakukan threading, ada 2 cara:
    - Membuat kelas yang mengextend Thread
    - Membuat kelas yang mengimplemen Runnable

CONTOH KELAS YANG *EXTEND* 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*.

CONTOH KELAS YANG *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();
...
```

EXTEND 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*

Bayangkan kalian diminta membuat sebuah game yang butuh 10 orang untuk dimulai. Saat diinvoke start(), Game menunggu 10 orang join game. Perhatikan kode berikut.

CONTOH WAIT & NOTIFY

```
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!");
  }
}
```

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:

CONTOH WAIT & 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();
    }
}
```

CONTOH WAIT & 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*.

CONTOH WAIT & NOTIFY

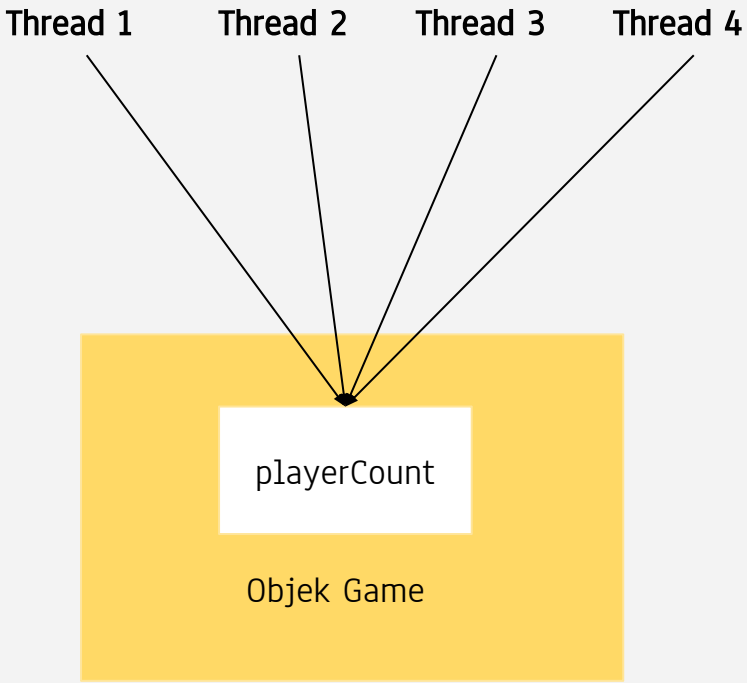
```
public class Game implements PlayerJoining.OnPlayerJoiningListener {
    private int playerCount;

    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!");
    }
}
```

SYNCHRONIZED

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();
}
```

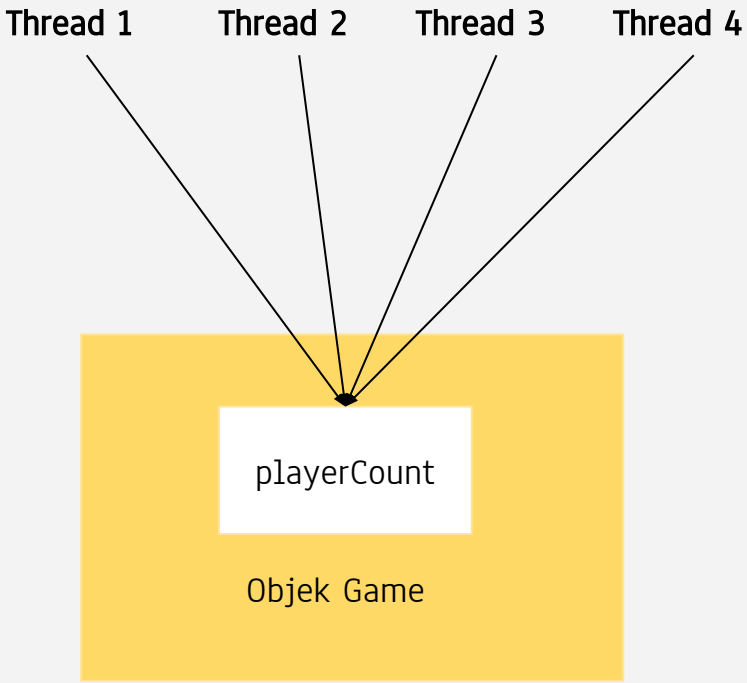
Sudah menemukan kesalahannya?



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

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

CONTOH WAIT & 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).