BAB VIII
Thread

A. Deflnisi Thread dan Multithreading

Thread adalah unit eksekusi terkecil dalam sebuah proses yang dapat berjalan secara independen. Setiap aplikasi yang dijalankan di komputer minimal memiliki satu thread utama yang disebut main thread.

Dalam Java, setiap program yang dieksekusi memiliki thread utama yang dimulai dari metode `main()`. Java menyediakan kelas `Thread` untuk membuat dan mengelola thread tambahan.

Multithreading adalah teknik pemrograman yang memungkinkan suatu program menjalankan beberapa thread secara bersamaan (*concurrent*) dalam satu proses. Dengan multithreading, sebuah aplikasi dapat melakukan banyak tugas sekaligus, sehingga meningkatkan efisiensi dan responsivitas.

Sebagai contoh, saat kalian *scrolling feed* pada aplikasi Instagram atau TikTok, sistem tetap dapat memuat konten baru di latar belakang, memutar video secara otomatis, serta menampilkan notifikasi secara *real-time* tanpa mengganggu interaksi utama pengguna. Dengan menggunakan multithreading, aplikasi dapat menjalankan beberapa tugas secara bersamaan, sehingga tetap responsif dan tidak mengalami lag, meskipun banyak proses berlangsung dalam waktu yang bersamaan.

B. Membuat Thread

Di Java, terdapat dua cara untuk membuat thread yaitu dengan mewarisi kelas `Thread` atau dengan mengimplementasikan interface `Runnable`.

1. Extends kelas Thread

Langkah-langkah untuk membuat thread dengan cara ini adalah sebagai berikut:

a. Buat sebuah kelas baru yang mewarisi kelas `Thread`

b. Override method `run()` untuk menentukan kode program yang akan dieksekusi oleh thread.

c. Buat sebuah objek dari kelas tersebut dan panggil method `start()` untuk memulai thread.

```java
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Running in a thread");
    }

    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
    }
}
```

Perlu diingat bahwa Java tidak mendukung pewarisan ganda, sehingga jika kelas kita sudah mewarisi kelas lain, kita tidak dapat menggunakan cara ini. Sebagai alternatif, kita bisa memanfaatkan interface `Runnable`.

2. **Implements interface Runnable**

   Langkah-langkah untuk membuat thread dengan cara ini adalah sebagai berikut:

   a. Buat sebuah kelas baru yang mengimplementasikan interface `Runnable`.

   b. Override method `run()` untuk menentukan kode program yang akan dieksekusi oleh thread.

   c. Buat objek dari kelas tersebut, lalu gunakan objek itu untuk membuat instance dari kelas `Thread` melalui konstruktor

   d. Panggil method `start()` pada objek thread untuk memulai thread.

```java
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Running in a thread");
    }

    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        thread.start();
    }
}
```

C. Menjalankan Thread di Java

Untuk menjalankan thread yang sudah dibuat, kita dapat menggunakan method `start()` pada objek thread tersebut. Method `start()` akan menjalankan thread secara *asynchronous*, sehingga thread tersebut dapat berjalan secara bersamaan dengan thread utama.

```
public class ThreadExample extends Thread {

    @Override
    public void run() {
        // Kode yang akan dijalankan di thread baru
        for (int i = 1; i <= 5; i++) {
            System.out.println("Thread Baru: " + i);
            try {
                Thread.sleep(1000); // Berhenti selama 1 detik
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        ThreadExample thread = new ThreadExample();

        // Menjalankan thread dengan method start()
        thread.start();

        // Kode di thread utama
        for (int i = 1; i <= 5; i++) {
            System.out.println("Thread Utama: " + i);
            try {
                Thread.sleep(1000); // Berhenti selama 1 detik
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Pada kode di atas, Ketika method `start()` dipanggil pada objek `ThreadExample`, thread baru dijalankan secara *asynchronous*, artinya ia berjalan tanpa menunggu thread utama selesai. Hal ini memungkinkan kedua thread (thread utama dan thread baru) berjalan secara bersamaan (*concurrent*).

Karena keduanya memakai `Thread.sleep(1000)`, tiap iterasi butuh 1 detik. Dengan multithreading, kedua thread berjalan paralel sehingga program selesai dalam 3 detik. Tanpa multithreading, eksekusi berurutan memakan waktu 6 detik. Multithreading mempercepat proses dan menunjukkan bagaimana dua thread dapat dijalankan secara bersamaan.

D. Menghentikan Thread di Java

Untuk menghentikan thread yang sedang berjalan, kita dapat menggunakan method stop(). Namun, method stop() tidak direkomendasikan karena dapat menyebabkan kesalahan pada data atau mengganggu kestabilan aplikasi. Sebagai gantinya, kita dapat menggunakan sebuah **flag boolean** untuk memberikan tanda pada thread untuk berhenti secara sukarela.

```
public class MyThread extends Thread {
    private boolean stop = false;

    @Override
    public void run() {
        // Jalankan selama thread belum diminta berhenti
        while (!stop) {
            System.out.println("Thread masih berjalan...");
            // Jeda 0.5 detik setiap perulangan
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println("Thread terganggu");
                break;
            }
        }
        System.out.println("Thread berhenti.");
    }

    public void stopThread() {
        stop = true;
    }

    public static void main(String[] args) {
        MyThread thread = new MyThread();
```

```
        // Jalankan thread
        thread.start();

        // Tunggu sampai 2 detik
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Hentikan thread setelah 2 detik
        thread.stopThread();
    }
}
```

E. Mengontrol Thread di Java

Terdapat beberapa method thread yang sering digunakan untuk mengontrol thread dalam java, yaitu:

a. **`start()`:** Memulai thread dan mengeksekusi method `run()`. Tidak boleh dipanggil lebih dari sekali pada thread yang sama, atau akan memicu `IllegalThreadStateException`.

b. **`run()`:** Berisi kode yang dijalankan oleh thread saat thread dimulai. Tidak menjalankan thread baru jika dipanggil langsung tanpa `start()`.

c. **`sleep(long millis)`:** Menunda eksekusi thread selama waktu tertentu (dalam milidetik). Harus ditangani dengan `InterruptedException`.

d. **`join()`:** Menunggu hingga thread selesai sebelum melanjutkan eksekusi. Bisa menerima argumen waktu maksimum menunggu (dalam milidetik).

Sebenarnya masih ada beberapa method lain seperti interrupt(), yield(), dll. Jadi, silahkan explore sendiri di link berikut:
https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

F. Masalah dalam Multithreading

Dalam multithreading, beberapa thread bisa berjalan bersamaan dan saling berbagi data. Namun, jika tidak dikelola dengan benar, bisa muncul masalah seperti:

1. **Race Condition**: Terjadi saat beberapa thread mengakses dan mengubah data yang sama secara bersamaan, sehingga hasil akhirnya bisa jadi tidak terduga.
2. **Deadlock**: Terjadi saat dua thread saling menunggu giliran untuk mengakses sumber daya (seperti data atau objek), tapi tidak ada yang mau mengalah, akhirnya keduanya berhenti total.

G. Sinkronisasi Thread

Sinkronisasi thread dalam Java adalah teknik yang digunakan untuk menghindari situasi di mana dua atau lebih thread saling bersaing untuk mengakses sumber daya bersama, seperti variabel atau objek, yang dapat menyebabkan *deadlock* atau *race condition*. Dalam Java, sinkronisasi thread dapat dicapai dengan menggunakan kata kunci `synchronized`.

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class SyncExample {
    public static void main(String[] args) {
        Counter counter = new Counter();

     // Membuat dua thread yang menambah nilai counter 1000 kali
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        t1.start();
        t2.start();

        // Menunggu kedua thread selesai
        try {
            t1.join();
            t2.join();
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Hasil akhir: " + counter.getCount());
    }
}
```

Kode di atas menggunakan dua thread untuk menambah nilai `count` sebanyak total 2000 kali. Method `increment()` diberi keyword `synchronized` agar hanya satu thread yang bisa mengaksesnya dalam satu waktu, sehingga mencegah *race condition*.

Jika `synchronized` tidak digunakan, kedua thread bisa saja menulis ke variabel `count` secara bersamaan, sehingga nilai akhirnya bisa kurang dari 2000 dan berubah-ubah setiap dijalankan.

Selain itu, method `join()` dipakai agar thread utama menunggu hingga kedua thread selesai menjalankan tugasnya sebelum mencetak hasil akhir nilai `count`.

H. Menjalankan Thread dengan Executor

Ketika kita ingin menjalankan beberapa tugas secara bersamaan (multithreading), biasanya kita membuat objek `Thread` satu per satu. Namun, pendekatan ini bisa merepotkan dan tidak efisien, terutama jika jumlah tugas banyak.

Untuk mengatasi hal ini, Java menyediakan `ExecutorService` melalui kelas `Executors`. Dengan menggunakan `Executor`, kita tidak perlu lagi membuat thread secara manual satu per satu. Kita cukup membuat sebuah thread pool, lalu menyerahkan tugas-tugas ke executor. Nantinya, executor akan secara otomatis membagikan tugas-tugas itu ke thread yang tersedia di dalam pool, dan mengatur eksekusinya secara paralel.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Executor {
    public static void main(String[] args) {
        // Membuat thread pool dengan 3 thread
        ExecutorService executor =
Executors.newFixedThreadPool(3);

        // Menjalankan 5 tugas berbeda
        executor.execute(() -> {
```

```
            System.out.println("Tugas 1: Kirim email - "
            + Thread.currentThread().getName());
            delay();
        });

        executor.execute(() -> {
            System.out.println("Tugas 2: Simpan ke database - "
            + Thread.currentThread().getName());
            delay();
        });

        executor.execute(() -> {
            System.out.println("Tugas 3: Cetak laporan - "
            + Thread.currentThread().getName());
            delay();
        });

        executor.execute(() -> {
            System.out.println("Tugas 4: Kirim notifikasi - "
            + Thread.currentThread().getName());
            delay();
        });

        executor.execute(() -> {
            System.out.println("Tugas 5: Buat backup - "
            + Thread.currentThread().getName());
            delay();
        });

        // Mematikan executor setelah semua tugas selesai
        executor.shutdown();
    }

     // Simulasi delay agar terlihat jeda antar tugas
    private static void delay() {
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Pada contoh di atas, kita membuat sebuah thread pool dengan kapasitas tiga thread menggunakan `Executors.newFixedThreadPool(3)`. Artinya, hanya ada tiga thread aktif yang dapat bekerja secara bersamaan. Kita kemudian mengirim lima tugas berbeda ke executor menggunakan `execute()`.

Karena hanya tersedia tiga thread, maka tiga tugas pertama akan langsung dijalankan secara paralel oleh ketiga thread tersebut, sedangkan dua tugas sisanya akan menunggu sampai ada thread yang selesai, lalu mengambil giliran.

Terakhir, kita memanggil `shutdown()` untuk memberitahu executor agar berhenti menerima tugas baru dan menutup semua thread setelah menyelesaikan tugas yang masih berjalan. Dengan pendekatan ini, manajemen thread menjadi jauh lebih sederhana, terstruktur, dan efisien.