

Kelompok 12

Thread Java

Anggota Kelompok

Irene Ully Havsa (0606101515 / Kelas B)

Mario Ray Mahardika (0606101686 / Kelas A)

Salman Salsabila (0606031566 / Kelas B)

Komentar umum

Secara umum, bab 12 membahas tentang konsep thread yang diimplementasikan dalam program berbahasa Java. Materi pembelajaran disampaikan secara sederhana dan mudah dimengerti. Namun, pembahasannya hanya dalam lingkup umum dan kurang mendalam.

Hubungan dengan bab sebelumnya dan sesudahnya

Dengan bab sebelumnya:

Bab sebelumnya membahas tentang konsep thread secara umum dan mendasar, sedangkan bab ini mengkhususkan dalam thread pada program Java.

Dengan bab sesudahnya:

Salah satu landasan bab ini adalah multithreading, sedangkan bahasan bab selanjutnya berkaitan dengan multiprograming. Keduanya sama-sama bertujuan untuk membuat komputer menjadi lebih produktif.

Komentar kelengkapan per bagian

12.1. Pendahuluan

Penjelasan mengenai thread pada bagian pendahuluan terlalu banyak mengutip dari bab sebelumnya, hanya saja dengan bahasa yang sedikit berbeda. Hal ini justru terasa sebagai pengulangan yang sia-sia dan membosankan, dan ada beberapa ketidak-sinkron-an dalam kata-kata penyampaian di kedua bab.

Selain itu pengantar tentang thread Java sendiri sangatlah kurang.

12.2. Status Thread

Secara keseluruhan penjelasannya cukup memuaskan dan jelas serta mudah dimengerti alurnya.

Gambar proses perubahan status ditampilkan dengan sederhana namun cukup menjelaskan.

Mungkin sebaiknya proses perubahan antar status dituliskan dengan lebih sistematis, misalnya proses pemanggilan method start() dijelaskan sebelum dituliskan poin status Runnable.

12.3. Pembentukan Thread

Pada subbab ini, perbedaan antara penggunaan extends dan implements belum begitu jelas, karena disampaikan tanpa pembagian topik. Kekurangan serta kelebihan antara keduanya pun tidak disampaikan.

Secara umum hanya perlu direvisi pembagian penulisan materinya.

12.4. Penggabungan Thread

Seperti yang telah di-posting sebelumnya, dalam bab ini tidak dijelaskan apakah pemanggilan join() harus dilakukan sebelum atau setelah start() dan apa efeknya jika dipanggil pada saat yang salah.

12.5. Pembatalan Thread

Secara keseluruhan, pembahasan materi sudah cukup baik. Namun masih perlu perbaikan dalam

penggunaan kata-kata dan kalimat.

12.6. JVM (Java Virtual Machine)

Penjelasan disampaikan dengan sangat terlalu singkat (hanya 7 baris saja...).

Sebaiknya lebih diperdalam dan lebih banyak penjelasan, atau digabung dengan subbab berikutnya saja.

12.7. Aplikasi Thread dalam Java

Penjelasan dalam bab ini pun sangat sangat singkat....

Dapat dikatakan bab ini hanya menampilkan salah satu contoh keluaran dari program pembatalan thread pada subbab 12.5.

Mungkin sebaiknya bab ini digabung dengan bab 12.5 tentang Pembatalan Thread.

12.8. Rangkuman

Rangkuman terlalu singkat dan banyak bagian yang 'hilang', sehingga menjadi tidak jelas dan kurang 'merangkum'.

Pembagian penulisannya juga kurang terstruktur.

Usulan kelengkapan

- a) Memperbaiki tata bahasa yang digunakan
- b) Memperbaiki struktur penulisan
- c) Memperjelas hal-hal yang disampaikan 'seadanya'
- d) Lebih memperdalam pembahasan
- e) Menambah pembahasan atau contoh aplikasi yang kurang memadai

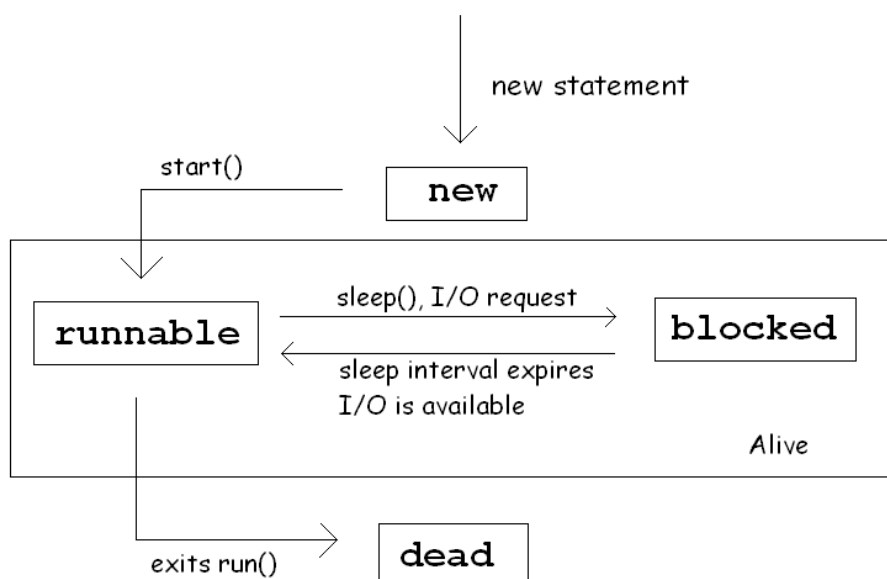
Bab 12. Thread Java

12.1. Pendahuluan

Pada bab sebelumnya telah dijelaskan secara umum pengertian serta kegunaan dari thread dan multithreading. Dalam kehidupan sehari-hari, penggunaan multithreading dapat kita lihat pada aplikasi *web*. Contohnya adalah sebuah *web browser* yang harus menampilkan sebuah halaman yang memuat banyak gambar. Pada program yang *single-threaded*, hanya ada satu *thread* untuk mengatur suatu gambar, lalu jika gambar itu telah ditampilkan, barulah gambar lain bisa diproses. Dengan *multithreading*, proses bisa dilakukan lebih cepat jika ada *thread* yang menampilkan gambar pertama, lalu *thread* lain untuk menampilkan gambar kedua, dan seterusnya, di mana *thread-thread* tersebut berjalan secara paralel.

Java adalah salah satu bahasa pemrograman yang mendukung penggunaan multithreading. Saat sebuah program Java dieksekusi, yaitu saat `main()` dijalankan, ada sebuah *thread* utama yang bekerja. Thread lain yang dibutuhkan dapat dibentuk dengan menuliskan kode program tertentu. Pada kondisi dan keperluan tertentu, thread pada program Java yang sedang dijalankan juga dapat digabung dengan thread utama ataupun dibatalkan. Keseluruhan *thread* dalam Java diatur oleh *Java Virtual Machine (JVM)* sehingga sulit untuk menentukan apakah *thread* Java berada di *user-level* atau *kernel-level*.

12.2. Status Thread



Gambar 12.1. Status Thread

Suatu *thread* bisa berada pada salah satu dari status berikut:

- **New.** *Thread* yang berada di status ini adalah objek dari kelas `Thread` yang baru dibuat, yaitu saat instansiasi objek dengan *statement new*. Saat *thread* berada di status *new*, belum ada sumber daya yang dialokasikan, sehingga *thread* belum bisa menjalankan perintah apapun.
 - ◊ Menjalankan *thread*. Agar *thread* bisa menjalankan tugasnya, method `start()` dari kelas `Thread` harus dipanggil. Ada dua hal yang terjadi saat pemanggilan method `start()`, yaitu alokasi memori untuk *thread* yang dibuat dan pemanggilan method `run()`.
- **Runnable.** Saat method `run()` dipanggil, status *thread* berubah menjadi *runnable*, artinya *thread* tersebut sudah memenuhi syarat untuk dijalankan oleh JVM. *Thread* yang sedang berjalan juga dikategorikan dalam status *runnable*.
 - ◊ *Blocking statement* dan interupsi M/K, yaitu kondisi yang menghalangi pengeksekusian *thread*. Ada berbagai jenis *blocking statement*. Yang pertama adalah pemanggilan method `sleep()`, suatu method yang menerima argumen bertipe *integer* dalam bentuk milisekon. Argumen tersebut menunjukkan seberapa lama *thread* akan "tidur". Selain `sleep()`, dulunya dikenal method `suspend()`, tetapi sudah disarankan untuk tidak digunakan lagi karena mengakibatkan terjadinya *deadlock*. Di samping *blocking statement*, adanya interupsi M/K juga dapat menyebabkan *thread* menjadi *blocked*.
- **Blocked.** Status *blocking* menunjukkan bahwa sebuah *thread* terhalang untuk menjalankan tugasnya, sehingga dapat dikatakan *thread* tersebut terhenti untuk sementara. *Thread* akan menjadi *runnable* kembali jika interval method `sleep()`-nya sudah berakhir, atau pemanggilan method `resume()` untuk mengakhiri method `suspend(,)` atau karena M/K sudah tersedia lagi.
 - ◊ Keluar dari method `run()`. Hal ini bisa terjadi karena *thread* tersebut memang telah menyelesaikan pekerjaannya di method `run()`, maupun karena adanya pembatalan *thread*.
- **Dead.** Setelah keluar dari method `run()`, *thread* akan berada dalam status *dead* dan menjadi tidak aktif lagi. Status jelas dari sebuah *thread* tidak dapat diketahui, tetapi method `isAlive()` mengembalikan nilai boolean untuk mengetahui apakah *thread* tersebut *dead* atau tidak.

12.3. Pembentukan *Thread*

Ada dua cara untuk membuat *thread* di program Java, yaitu *extends* kelas `Thread` dan *implements* interface `Runnable`.

Interface `Runnable` didefinisikan sebagai berikut:

```
public interface Runnable
{
    public abstract void run();
}
```

a. Extends kelas `Thread`

Kelas `Thread` secara implisit juga meng-*implements interface* `Runnable`. Oleh karena itu,

setiap kelas yang diturunkan dari kelas `Thread` juga harus mendefinisikan method `run()`. Berikut ini adalah contoh kelas yang menggunakan cara pertama untuk membuat *thread*, yaitu dengan meng-*extends* kelas `Thread`.

```
class CobaThread1 extends Thread
{
    public void run()
    {
        for (int ii = 0; ii<4; ii++){
            System.out.println("Ini CobaThread1");
            Test.istirohat(11);
        }
    }
}
```

Konsep pewarisan dalam Java tidak mendukung *multiple inheritance*. Hal ini menjadi kekurangan dari penggunaan metode *extends*. Jika sebuah kelas sudah meng-*extends* suatu kelas lain, maka kelas tersebut tidak lagi bisa meng-*extends* kelas `Thread`.

b. Implements interface Runnable

Adanya kekurangan yang cukup fatal pada metode *extends*, maka cara kedua, yaitu meng-*implements interface* `Runnable`, lebih umum digunakan, karena kita bisa meng-*implements* dari banyak kelas sekaligus.

```
class CobaThread2 implements Runnable
{
    public void run()
    {
        for(int ii = 0; ii<4; ii++){
            System.out.println("Ini CobaThread2");
            Test.istirohat(7);
        }
    }
}

public class Test
{
    public static void main (String[] args)
    {
        Thread t1 = new CobaThread1();
        Thread t2 = new Thread (new CobaThread2());
        t1.start();
        t2.start();

        for (int ii = 0; ii<8; ii++){
            System.out.println("Thread UTAMA");
            istirohat(5);
        }
    }
    public static void istirohat(int tunda)
    {
        try{
            Thread.sleep(tunda*100);
        } catch (InterruptedException e) {}
    }
}
```

Pada bagian awal `main()`, terjadi instansiasi objek dari kelas `CobaThread1` dan `CobaThread2`, yaitu `t1` dan `t2`. Perbedaan cara penginstansian objek ini terletak pada perbedaan akses yang dimiliki oleh kelas-kelas tersebut. Supaya *thread* bisa bekerja, method `start()` dari kelas `Thread` harus dipanggil. Kelas `CobaThread1` memiliki akses ke method-method yang ada di kelas `Thread` karena merupakan kelas yang diturunkan langsung dari kelas `Thread`. Namun, tidak demikian halnya dengan kelas `CobaThread2`. Oleh karena itu, kita harus tetap membuat objek dari kelas `Thread` yang menerima argumen objek `CobaThread2` pada *constructor*-nya, barulah `start()` bisa diakses. Hal ini ditunjukkan dengan *statement* `Thread t2 = new Thread (new CobaThread2())`.

Jadi, ketika terjadi pemanggilan method `start()`, *thread* yang dibuat akan langsung mengerjakan baris-baris perintah yang ada di method `run()`. Jika `run()` dipanggil secara langsung tanpa melalui `start()`, perintah yang ada di dalam method `run()` tersebut akan tetap dikerjakan, hanya saja yang mengerjakannya bukanlah *thread* yang dibuat tadi, melainkan *thread* utama.

12.4. Penggabungan *Thread*

Tujuan *multithreading* adalah agar *thread-thread* melakukan pekerjaan secara paralel sehingga program dapat berjalan dengan lebih baik. *Thread* tambahan yang dibuat akan berjalan secara terpisah dari *thread* yang membuatnya. Namun, ada keadaan tertentu di mana *thread* utama perlu menunggu sampai *thread* yang dibuatnya itu menyelesaikan tugasnya. Misalnya saja, untuk bisa mengerjakan instruksi selanjutnya, *thread* utama membutuhkan hasil penghitungan yang dilakukan oleh *thread* anak. Pada keadaan seperti ini, *thread* utama bisa menunggu selesainya pekerjaan *thread* anak dengan pemanggilan method `join()`.

Contohnya, dalam suatu program, *thread* utama membuat sebuah *thread* tambahan bernama `t1`.

```
try{
    t1.join();
} catch (InterruptedException ie) {};
```

Kode di atas menunjukkan bahwa *thread* utama akan menunggu sampai *thread* `t1` menyelesaikan tugasnya, yaitu sampai method `run()` dari `t1` *terminate*, baru melanjutkan tugasnya sendiri. Pemanggilan method `join()` harus diletakkan dalam suatu blok `try-catch` karena jika pemanggilan tersebut terjadi ketika *thread* utama sedang diinterupsi oleh *thread* lain, maka `join()` akan melempar `InterruptedException`. `InterruptedException` akan mengakibatkan terminasi *thread* yang sedang berada dalam status *blocked*.

12.5. Pembatalan *Thread*

Pembatalan *thread* adalah menterminasi sebuah *thread* sebelum tugasnya selesai. *Thread* yang akan dibatalkan, atau biasa disebut *target thread*, dapat dibatalkan dengan dua cara, yaitu *asynchronous cancellation* dan *deferred cancellation*. Pada *asynchronous*

cancellation, sebuah *thread* langsung menterminasi *target thread*. Sedangkan pada *deferred cancellation*, *target thread* secara berkala memeriksa apakah ia harus *terminate* sehingga dapat memilih saat yang aman untuk *terminate*.

Pada *thread* Java, *asynchronous cancellation* dilakukan dengan pemanggilan method `stop()`. Akan tetapi, method ini sudah di-*deprecated* karena terbukti tidak aman karena dapat mengakibatkan terjadinya *exception* `ThreadDeath`. `ThreadDeath` dapat mematikan *thread-thread* secara diam-diam, sehingga *user* mungkin saja tidak mendapat peringatan bahwa programnya tidak berjalan dengan benar.

Cara yang lebih aman untuk membatalkan *thread* Java adalah dengan *deferred cancellation*. Pembatalan ini dapat dilakukan dengan pemanggilan method `interrupt()`, yang akan mengeset status interupsi pada *target thread*. Sementara itu, *target thread* dapat memeriksa status interupsi-nya dengan method `isInterrupted()`.

```
class CobaThread3 implements Runnable
{
    public void run(){
        while (true){
            System.out.println("saya thread CobaThread3");
            (Thread.currentThread().isInterrupted()) //cek status
            break;
        }
    }
}
```

Suatu *thread* dari kelas `CobaThread3` dapat diinterupsi dengan kode berikut:

```
Thread targetThread = new Thread (new CobaThread3());
targetThread.start();
...
targetThread.interrupt(); //set status interupsi
```

Ketika *thread* `targetThread` berada pada `start()`, *thread* tersebut akan terus me-*loop* pada method `run()` dan melakukan pengecekan status interupsi melalui method `isInterrupted()`. Status interupsinya sendiri hanya akan di-set ketika pemanggilan method `interrupt()`, yang ditunjukkan dengan *statement* `targetThread.interrupt();`. Setelah status interupsi di-set, ketika pengecekan status interupsi selanjutnya pada method `run()`, `isInterrupted()` akan mengembalikan nilai boolean `true`, sehingga `targetThread` akan keluar dari method `run()`-nya melalui *statement* `break` dan *terminate*.

Selain melalui `isInterrupted()`, pengecekan status interupsi dapat dilakukan dengan method `interrupted()`. Perbedaan kedua method ini adalah `isInterrupted()` akan mempertahankan status interupsi, sedangkan pada `interrupted()`, status interupsi akan di-*clear*.

Thread yang statusnya sedang *blocked* karena melakukan operasi M/K menggunakan *package* `java.io` tidak dapat memeriksa status interupsinya sebelum operasi M/K itu selesai. Namun, melalui Java 1.4 diperkenalkan *package* `java.nio` yang mendukung interupsi *thread* yang sedang melakukan operasi M/K

12.6. JVM

Di dalam JVM, setiap *thread* mempunyai *stack* sendiri-sendiri, yang berisi data yang tidak dapat diakses oleh *thread* lain, termasuk variabel lokal, parameter, dan nilai balik dari setiap method yang dijalankan oleh *thread* tersebut. Data pada *stack* terbatas pada *primitive type* dan *object reference*. Di JVM, tidak dimungkinkan untuk meletakkan objek yang sesungguhnya di *stack*. Semua obyek berada di *heap*.

Hanya ada satu *heap* di dalam JVM, dan seluruh *thread* membaginya. *Heap* hanya berisi obyek-obyek. Tidak ada cara untuk meletakkan *primitive type* maupun *object reference* di *heap*, keduanya harus menjadi bagian dari objek. Array berada di *heap*, termasuk array yang elemennya berupa *primitive type*, karena di Java, array juga merupakan objek.

Di samping *stack* dan *heap*, tempat lain di mana data boleh berada adalah *method area*, yang berisi variabel kelas (atau *static*) yang digunakan oleh program. *Method area* mirip dengan *stack* di mana isinya hanya *primitive type* dan *object reference*. Perbedaannya, variabel kelas pada *method area* dibagi oleh semua *thread*.

Jika banyak *thread* perlu menggunakan objek atau variabel kelas yang sama secara bersamaan, akses terhadap data harus diatur dengan benar. Jika tidak, program akan memberikan hasil yang tidak terduga.

Untuk mengatur akses tersebut, JVM menghubungkan sebuah “kunci” dengan setiap objek dan kelas. Kunci ini seperti hak istimewa di mana hanya satu *thread* yang bisa mengakses pada satu waktu. Jika suatu *thread* ingin mengunci objek atau kelas tertentu, maka ia akan meminta pada JVM. Pada suatu waktu setelah ia meminta (mungkin langsung, nanti, atau tidak akan pernah), JVM memberikan kunci pada *thread* tersebut. Ketika *thread* tersebut tidak memerlukannya lagi, kunci tersebut dikembalikan pada JVM. Jika ada *thread* lain yang telah meminta kunci tersebut, JVM akan memberikannya pada *thread* tersebut.

Penguncian kelas sebenarnya diimplementasikan sebagai objek kunci. Ketika JVM me-load sebuah file .class, maka instance dari kelas `java.lang.Class` akan dibuat. Ketika sebuah kelas dikunci, yang dikunci sebenarnya objek `Class` dari kelas tersebut.

Thread tidak perlu memiliki kunci untuk mengakses variabel *instance* atau kelas. Namun, jika suatu *thread* melakukan penguncian, *thread* lain tidak dapat mengakses data yang dikunci sampai *thread* yang memiliki kunci tersebut membukanya.

12.7. Aplikasi *Thread* dalam Java

Dalam ilustrasi program yang ada pada subbab Pembatalan *Thread*, kita tidak dapat mengetahui *thread* yang mana yang akan terlebih dahulu mengerjakan tugasnya. Hal ini terjadi karena ada dua *thread* yang berjalan secara paralel, yaitu *thread* utama dan *thread* t1. Artinya, keluaran dari program ini bisa bervariasi. Salah satu kemungkinan keluaran program ini adalah sebagai berikut:

```
Thread UTAMA
Ini CobaThread1
Ini CobaThread2
Thread UTAMA
Ini CobaThread2
Thread UTAMA
Ini CobaThread1
Ini CobaThread2
Thread UTAMA
Thread UTAMA
Ini CobaThread2
```

```
Ini CobaThread1
Thread UTAMA
Thread UTAMA
Ini CobaThread1
Thread UTAMA
```

12.8. Rangkuman

Setiap program Java memiliki paling sedikit satu *thread*, yang otomatis terbentuk saat dieksekusi. Untuk kebutuhan tertentu, bahasa pemrograman Java memungkinkan adanya pembuatan dan manajemen *thread* tambahan oleh JVM (Java Virtual Machine).

Sebuah *thread* bisa berada di salah satu dari 4 status, yaitu ***new***, ***runnable***, ***blocked***, dan ***dead***. Ada dua cara untuk membuat *thread* dalam Java, yaitu dengan meng-*extends* kelas `Thread` atau dengan meng-*implements interface* `Runnable`.

Dalam beberapa kondisi, *thread* yang dibuat dapat digabungkan dengan *parent thread*-nya. Penggabungan ini menggunakan method `join()`, yang berfungsi agar suatu *thread* induk menunggu *thread* yang dibuatnya selesai menjalankan tugasnya, baru mulai mengeksekusi perintah selanjutnya.

Pembatalan *thread* secara *asynchronous* dilakukan dengan pemanggilan method `stop()`. Akan tetapi, cara ini terbukti tidak aman, sehingga untuk menterminasi *thread* digunakanlah *deferred cancellation*. Pembatalan dilakukan dengan pemanggilan method `interrupt()` untuk mengeset status interupsi, serta `isInterrupted()` atau `interrupted()` untuk memeriksa status interupsi tersebut.

Program Java dapat dijalankan di berbagai *platform* selama *platform* tersebut mendukung JVM. Pemetaan *thread* Java ke *host operating system* tergantung pada implementasi JVM di sistem operasi tersebut.

12.9. Rujukan

- [Lewis1998] John Lewis dan William Loftus. 1998. *Java Software Solutions Foundation Of Program Design*. First Edition. Addison Wesley.
- [Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.
- [Tanenbaum1997] Andrew Tanenbaum dan Albert Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.
- [WEBJava2007] Java 2 Platform SE v1.3.1. 2007. *Java 2 Platform SE v1.3.1: Class Thread* – <http://java.sun.com/j2se/1.3/docs/api/java/lang/Thread.html>. Diakses 27 Februari 2007.
- [WEBJTPD2007] Java Thread Primitive Deprecation. 2007. *Java Thread Primitive Deprecation* – <http://java.sun.com/j2se/1.3/docs/guide/misc/threadPrimitiveDeprecation.html>. Diakses 27 Februari 2007.