**Java Thread**

## **Pendahuluan**

**Sebelumnya harus sudah paham materi terkait** :

* Java Dasar
* Java Object Oriented Programming
* Java Generic
* Java Collection & Stream
* Java Lambda
* Apache Maven

**Agenda**

* Pengenalan Concurency
* Thread
* Synchronized
* Lock
* Deadlock
* Race Condition
* Threadpool
* ExcutorService
* Dan lain – lain

## **Pengenalan Concurrency**

**Sejarah Concurency**

* Dahulu, computer hanya menjalankan sebuah program pada satu waktu
* Karena hanya bisa menjalankan satu program pada satu waktu, hal ini tidak efisien dan memakan wakatu lama karena hanya bisa mengerjakan tugas pada satu waktu
* Semakin kesini, sistem operasi untuk computer semakain berkembang, sekarang sistem operasi bisa menjalankan program secara bersamaan dalam proses yang berbeda-beda, terisolasi dan saling independent antar program

**Sejarah Thread**

* Program biasanya berjalan dalam sebuah proses, dan proses akan memiliki resource yang independen dengan proses lain
* Sekarang, sistem operasi tidak hanya bisa menjalankan multiple proses, namun dalam proses kitab isa menjalankan banyak pekerjaan sekaligus, atau bisa dibilang proses ringan atau lebih dikenal dengan nama Thread
* Thread membuat proses aplikasi bisa berjalan tidak harus selalu sequential, kitab isa membuat proses apliaksi berjlan menjadi asynchronous atau parallel

**Era Multicore**

* Sekarang kita sudah ada di zaman multicore, dimanan smartphone pun sudah menggunakan multicore
* Multicore sangat menguntungkan kita karena bisa membuat aplikasi yang bisa menjalankan proses dan thread secara bersamaan

**Concurrency vs Parallel**

* Kadang banyak yang bingung dengan concurrency dan parallel, sebenarnya kita tidak perlu terlalu memusingkan hal ini
* Karena saat ini, kita pasti akan menggunakan keduanya ketika membuat aplikasi
* Concurrency artinya mengerjakan beberapa pekerjaan satu persatu pada satu waktu
* Parallel artinya mengerjakan beberapa pekerjaan sekaligus pada satu waktu

**Conntoh Concurrency dan Parallel**

* Browser adalah apliaksi yang concurrent dan parallel
* Browser akan melakukan proses concurrency ketika membuka web, browser akan melakukan http request, lalu mendowload semua file web (html, css, js ) lalu merender dalam bentuk tampilan web
* Browser akan melakukan proses parallel, ketika membuka beberapa tab web, dan juga sambal download beberapa file, dan menonton video dari web streaming

**Synchronous vs Asynchronous**

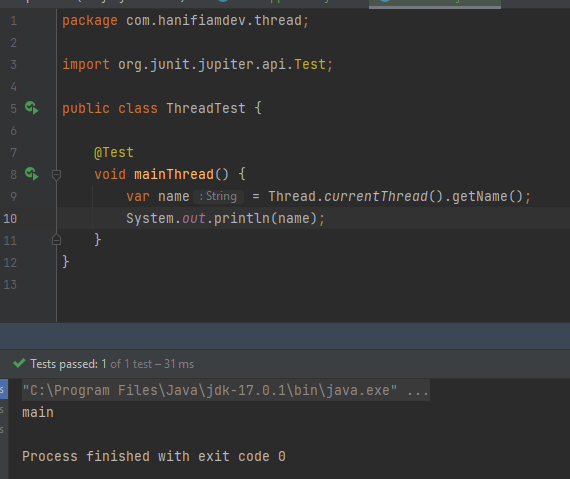
* Saat membuat aplikasi yang concurrent atau parallel, kadang kita sering menemui istilah synchronous dan asynchronous
* Tidak perlu bingung dengan istilah tersebut, secara sederhana
* Sysnchronous adalah ketika kode program kita berjalan secara sequential, dan semua tahapan ditunggu sampai prosesnya selesai baru akan dieksekusi ke tahapan selanjutnya
* Sedangkan, Asyschronous artinya ketika kode program kita berjalan dan kita tidak perlu menunggu eksekusi kode tersebut selesai, kita bisa lanjutkan ke tahapan kode program selanjutnya

## **Thread**

* Di java, implementasi Concurrency dan Parallel dapat menggunakan Thread
* Thread direpresentasikan oleh class bernama Thread di package.lang

**Thread Utama**

* Secara default, saat sebuah aplikasi Java berjalan, minimal aka ada satu thread yang bezrjalan
* Dalam apliaksi Java biasa, biasanya kode kode program kita akan berjalan di dalam thread yang bernama main
* Bahkan di Unit Test pun, memiliki thread sendiri



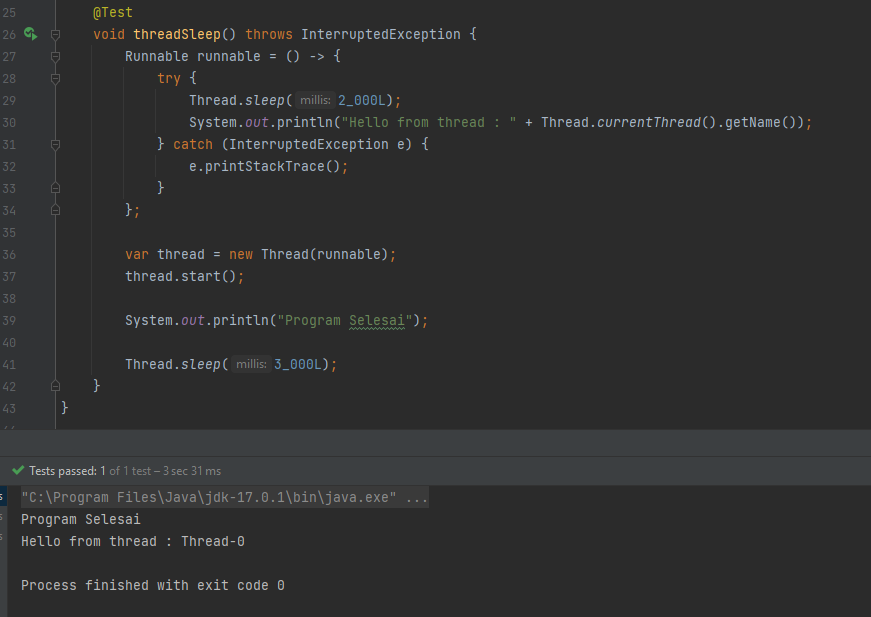
**Membuat Thread**

* Thread merupakan proses ringan, membuat Thread bukan berarti kita melakukan pekerjaan
* Untuk membuat pekerjaan, kita perlu membuat object dari interface Runnable, selanjutnya object Runnable tersebut bis akita berikan ke Thread untuk dijalankan
* Saat Thread berjalan, dia akan berjalan secara asynchronous, artinya dia akan berjalan sendiri, dan kode program kita akan berlanjut ke kode program selanjutnya
* Untuk menjalankan Thread, kitab isa memanggil function start() milik Thread



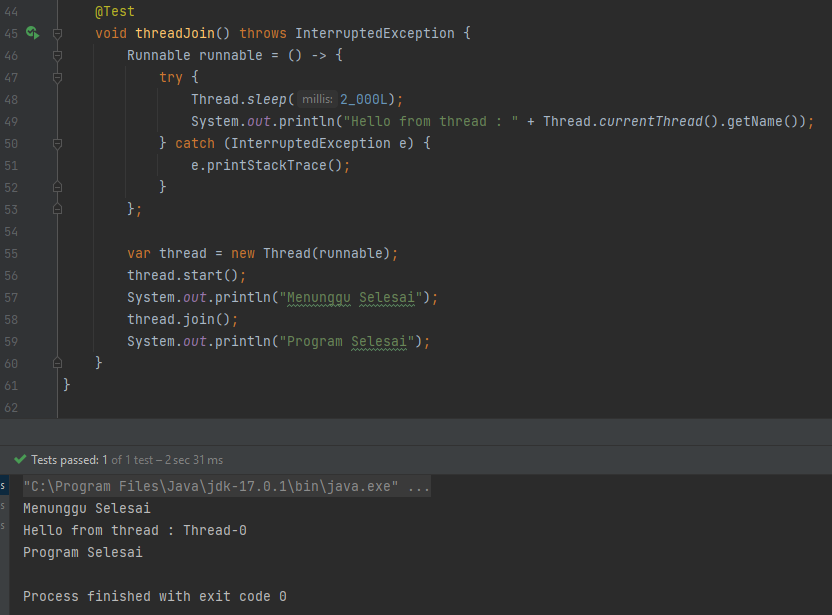
**Thread Sleep**

* Pada saat proses development, kadang kita butuh melakukan simulasi proses yang berjalan dalam waktu tertentu
* Untuk melakukan hal ini, bisa memanfaatkan fitur Thread Slwwp yang terdapat di Java
* Dengan menggunakan Thread Sleep, kitab isa membuat sebuah thread tertidur dan berhenti dalam waktu yang kita tentukan
* Untuk melakukan hal ini, kitab isa memanggil static method sleep() di class Thread, maka secara otomatis Thread saat itu dimana memanggil kode sleep() akan tertidur sesuai dengan waktu millisecond yang sudah kita masukkan dalam parameter
* Namun perlu di perhatikan, method sleep bisa menyebabkan eror InterruptedException



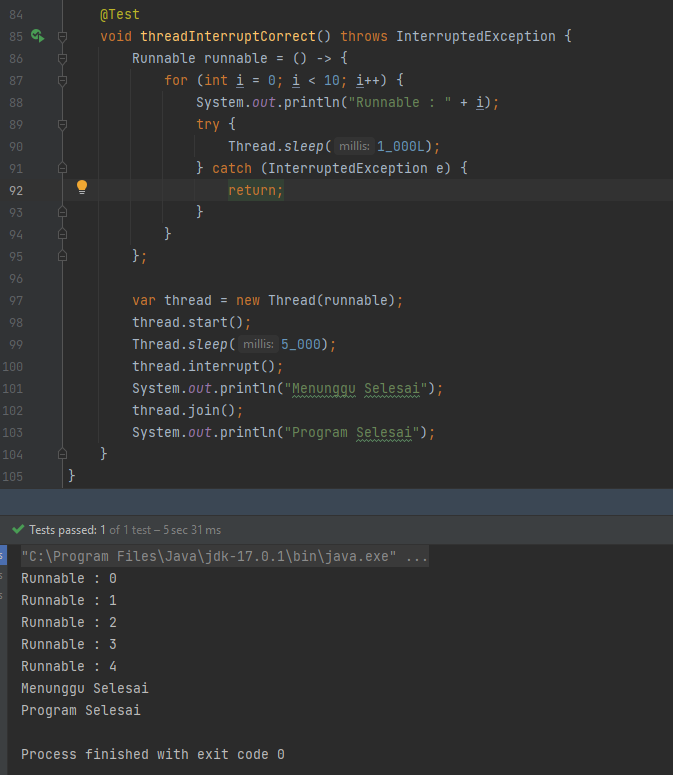
**Thread Join**

* Kadang kadang kita menunggu sampai sebuah thread selesai tugasnya
* Untuk melakukan hal tersebut, kitab isa memanggil method join milik thread yang akan kita tungggu

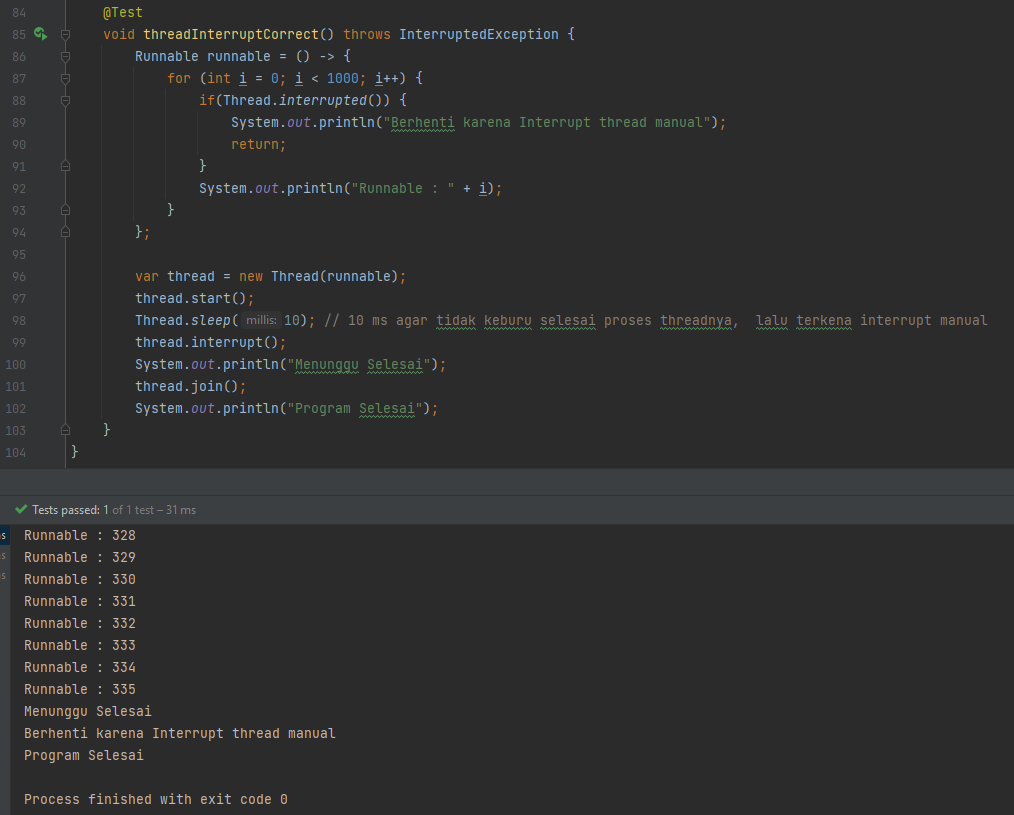


**Thread Interrupt**

* Interrupt merupakan mengirim sinyal ke thread bahwa thread tersebut harus berhenti melakukan pekerjaannya saat ini
* Untuk melakukan interrupt, kita bisa menggunakan method interrupt() pada thread
* Saat kita memanggil method interrupt(), secara otomatis Thread.Interrupted() akan bernilai true
* Perlu diingat, kode program kita pada Runnable harus melakukan pengecekan interrupted, jika tidak, sinyal interrupt tidak ada gunanya



**Interrupt Thread Manual**



**Thread Name**

* Secara default thread di java memiliki nama, ini sangat cocok untuk proses debugging ketika terjadi masalah
* Thread name secara default akan menggunakan nama Thread-{counter}
* Namun kita juga bisa mengubahnya dengan menggunakan method setName(name), dan getName() untuk mendapatkan nama thread nya



**Thread State**

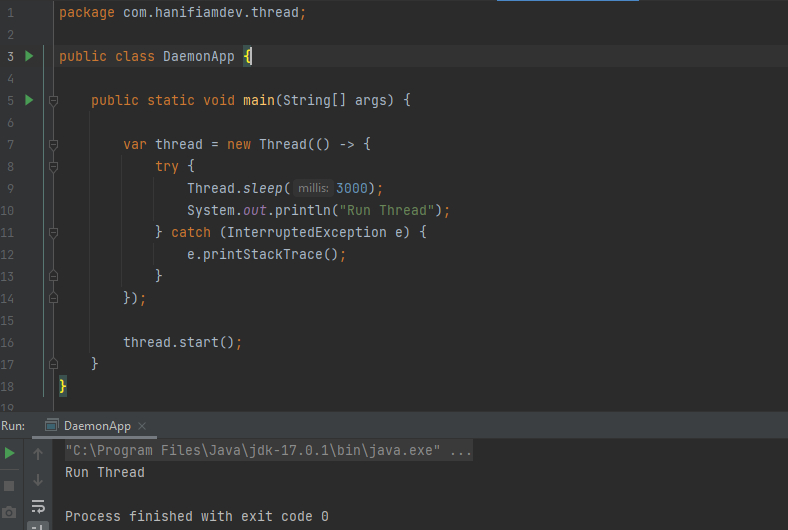
* Thread di Java memiliki state, state yaitu informasi state sebuah thread
* State digunakan hanya untuk mendapatkan informasi
* State akan berubah statusnya sesuai dengan apa yang terjadi di thread
* Untuk mendapatkan data state, kita bisa menggunakan method getState() dan akan mengembalikan enum State



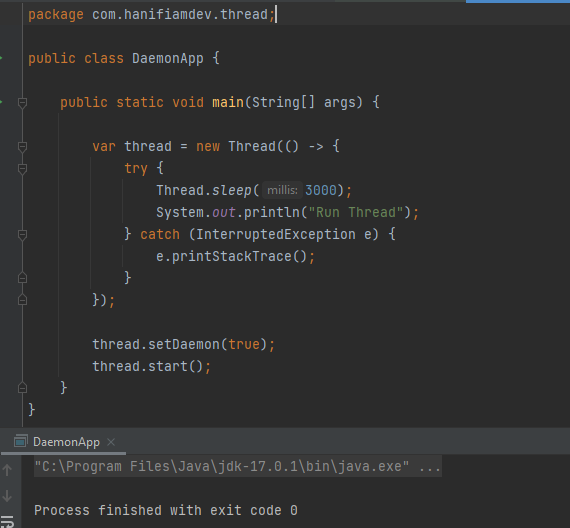
**Thread Daemon**

* Secara default, saat kita membuat thread, thread tersebut disebut user thread
* Java (bukan JUnit) secara default akan selalu menunggu semua user thread selesai sebelum program berhenti
* Jika kita mengubah thread menjadi daemon thread, menggunakan setDaemon(true), maka secara otomatis thread tersebut menjadi daemon thread
* Namun jika menghentikan program Java menggunakan System.exit(), maka user thread pun akan otomatis terhenti

Kondisi default / user thred ( akan ditunggu )



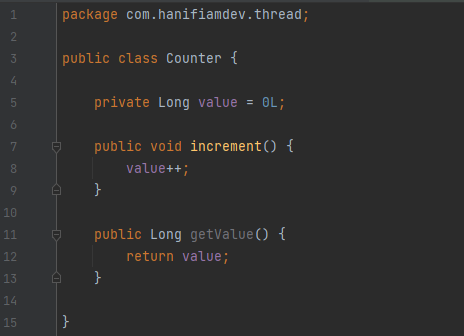
daemon thread( ini cocok untuk background proses atau tidak perlu ditunggu balikanya)



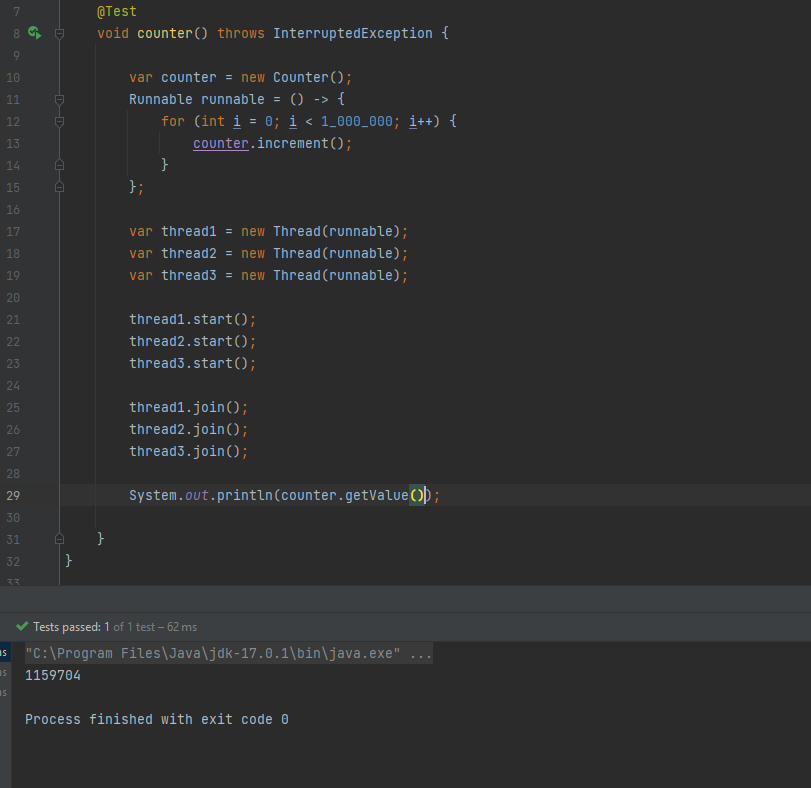
## **Race Condition**

* Salah satu problem yang sering sekali terjadi dalam aplikasi concurrent dan parallel adalah race condition
* Race condition merupakan keadaan ketika sebuah data diubah secara berbarengan oleh beberapa thread yang menyebabkan hasil akhir yang tidak sesuai dengan yang kita inginkan

Buat class Counter



Lakukan Test untuk menjalankan multiple Thread dengan mengakses objek counter



*Note : banyak data yang seakan hilang karena terjadi rece condition.*

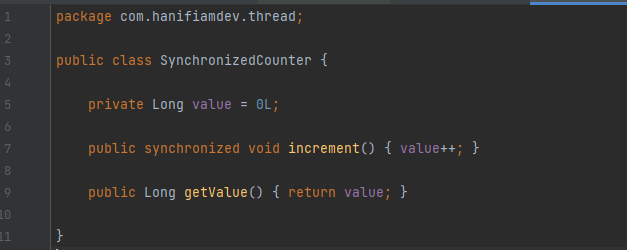
## **Synchronization**

* Masalah race condition sebelumnya yang terjadi di Java bisa diselesaikan dengan Synchronization
* Synchronization merupakan fitur dimana kita memaksa kode program hanya boleh diakses dan dieksekusi oleh satu thread saja
* Hal ini menyebabkan thread yang lain yang mengakses kode program tersebut harus menunggu thread yang lebih dahulu mengakses, sehingga proses Synchronization akan lebih lambat, namun proses Synchronization lebih aman karena tidak akan terjadi race condition

**Synchronized Method**

* Di Java, terdapat dua jenis synchronization, yaitu synchronized method dan synchronized statement
* Synchronized method method merupakan synchronization paling mudah, karena kita hanya perlu menambah kata kunci synchronized pada method yang ingin kita set sebagai synchronization
* Dengan begitu, secara otomatis method tersebut hanya bisa diakses oleh satu thread pada satu waktu

Buat class Counter SynchronizedCounter yang menambah kata kunci synchronized pada method increment



Lakukan Test hasilnya sesuai yaitu 3 juta, hal ini karena tidak terjadi race condition



**Intrinsic Lock**

* Synchronization di Java sebenarnya bekerja menggunakan lock
* Ketika kita melakukan synchronized method, secara otomatis Java akan membuat intrinsic lock atau monitor lock
* Ketika synchronized method dipanggil oleh thread, thread akan melakukan mencoba mendapatkan intrinsic lock, setelah method selesai ( sukses ataupun error ), maka thread akan mengembalikan instrinsic lock
* Semua itu terjadi sejari secara otomatis di synchronized method

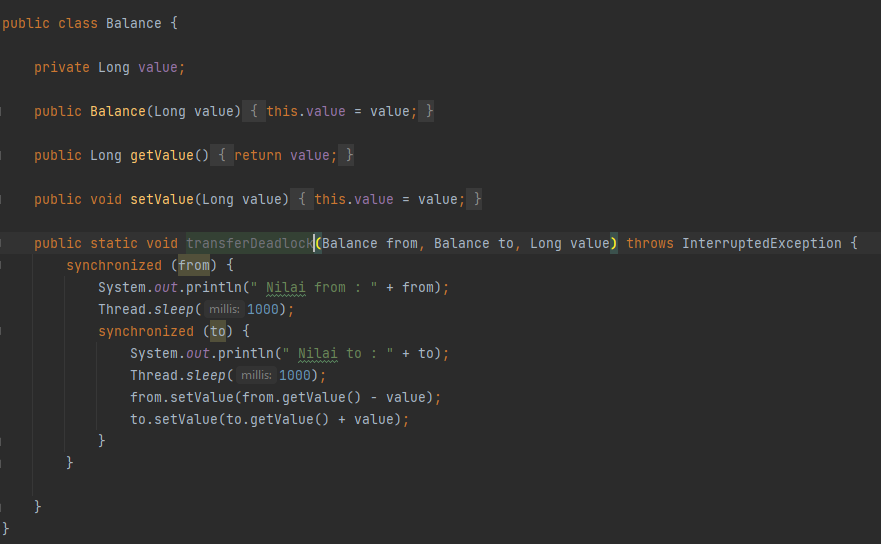
**Synchronized Statement**

* Saat kita menggunakan synchronized method secara otomatis seluruh method akan tersynchronization
* Kadang, misal hanya ingin melakukan synchronized pada bagian kode tertentu saja
* Untuk melakukan hal tersesbut, kita bisa menggunakan synchronized statement
* Namun ketika kita menggunakan synchronized statement, kita harus menentukan object intrinsic lock sendiri

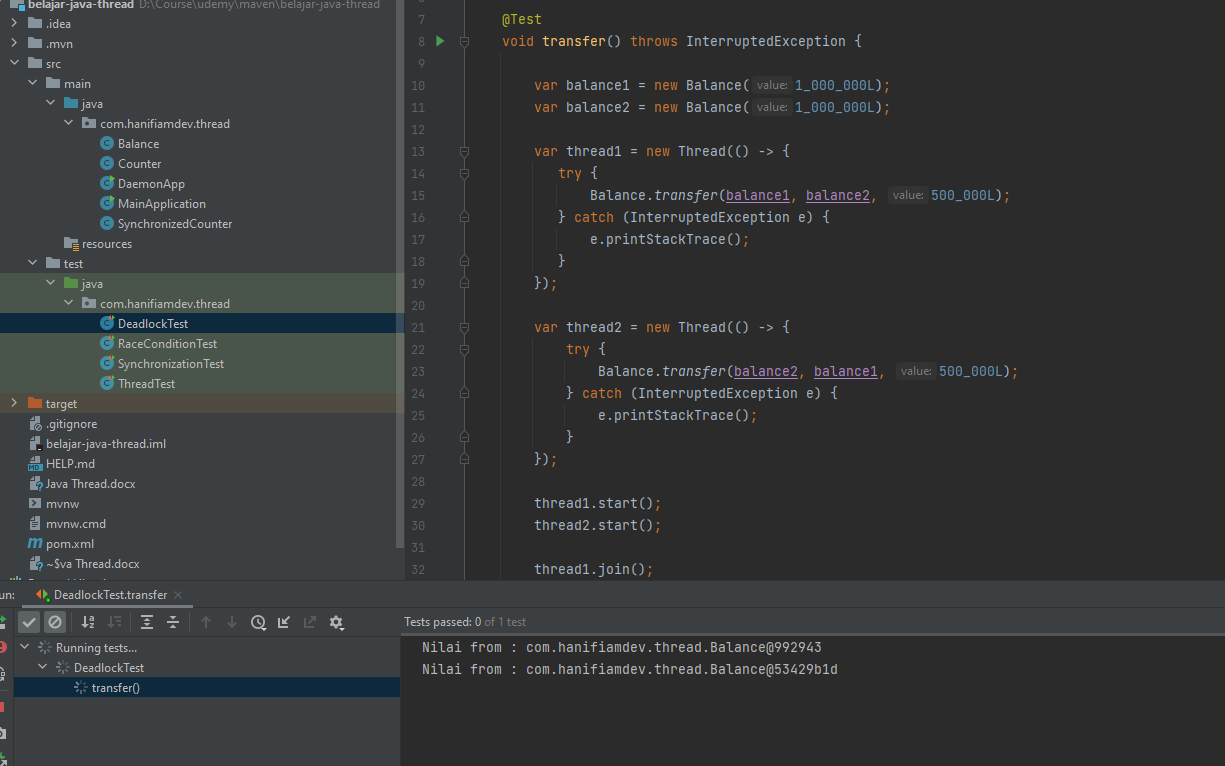
## **DeadLock**

* Race condition sangat mudah diselesaikan dengan synchronization dan lock, namun masalah lain bisa mengintai jika kita selalu melakukan synchronization, yaitu Deanlock
* Deadlock merupakan kondisi dimana beberapa thread saling menunggu satu sama lain karena biasanya terjadi ketika thread tersebut melakukan lock dan menunggu lock lain dari thread lain dan ternyata thread tersebut juga menunggu lock lain
* Karena saling menunggu, akhirnya terjadi deadlock, keadaan dimana semua thread tidak berjalan karena hanya menunggu lock

Buat class Balance untuk simulasi deadlock



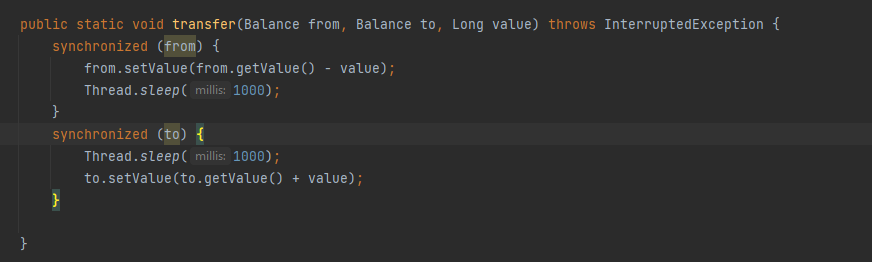
Test deadlock, akan mengambil kasus dimana thread satu akan melakukan lock terhadap thread 2, begitu sebaliknya.

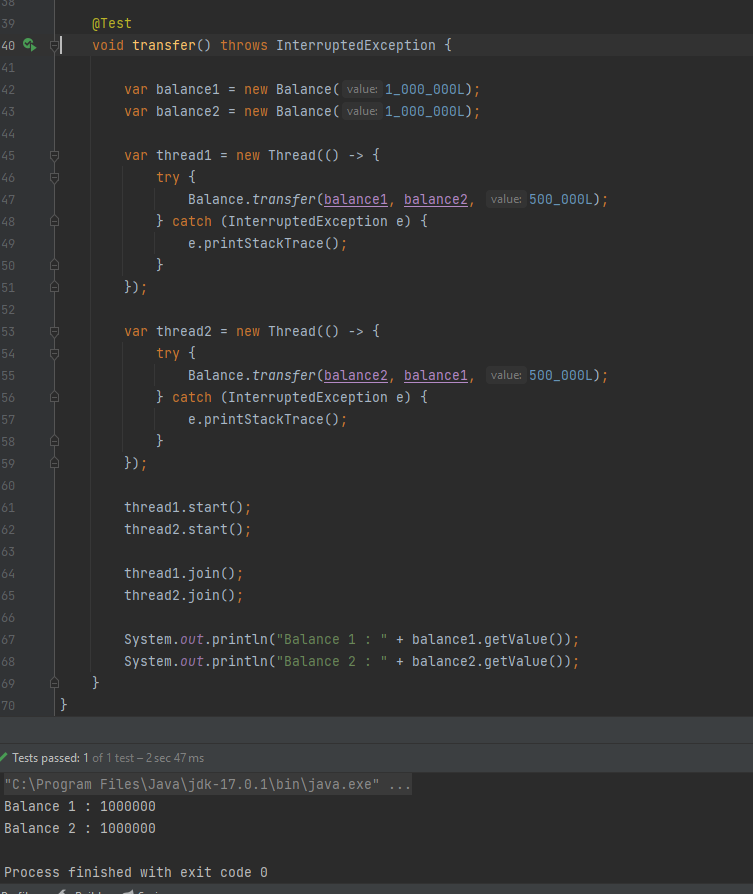


**Cara Menangani Deadlock**

* Sayangnya tidak ada cara menyelesaikan masalah deadlock secara otomatis di Java
* Masalah deadlock harus diselesaikan sendiri oleh programmer yang membuat kode program nya

Perbaikan deadlock salah satunya dengan merubah logic program dalam penggunanaan synchronized

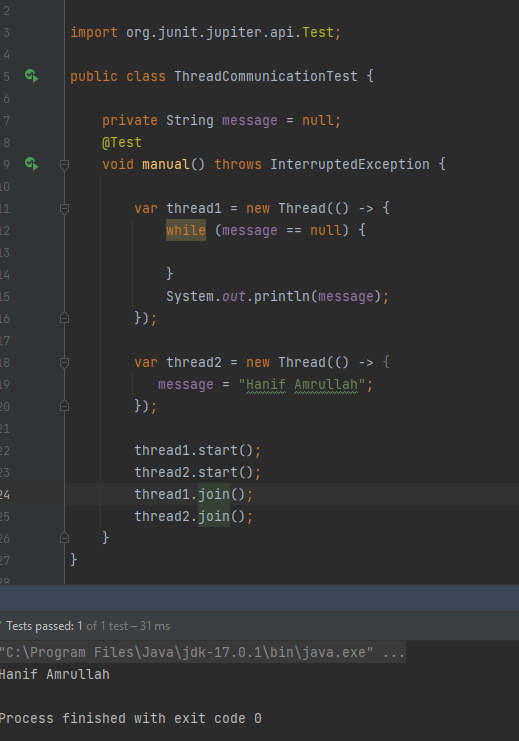




## **Thread Communication**

* Dalam multithreaded programming, kadang sudah biasa ketika sebuah thread perlu menunggu thread lain menyelesaikan tugas tertentu, baru thread tersebut melakukan tugasnya
* Sayangnya tidak ada cara otomatis komunikasi antar thread secaar langsung
* Oleh karena itu, programmer harus melakukannya secara manual untuk komunikasi antar thread

Sharing variable ( cara ini tidak direkomendasikan )



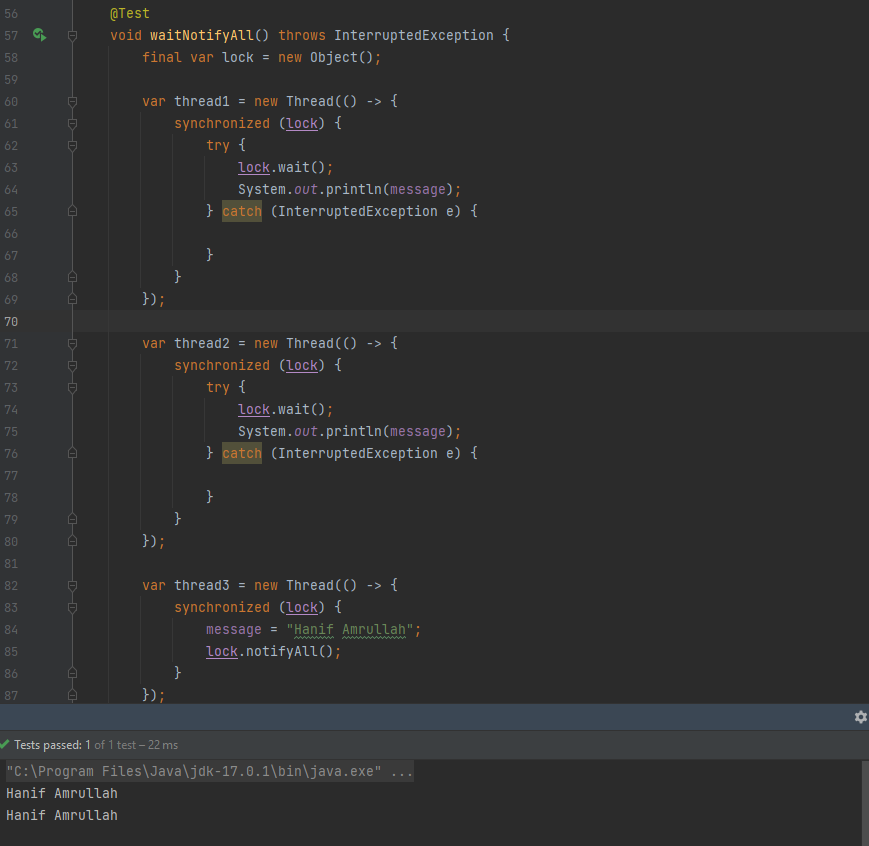
**Wait dan Notify**

* Menggunakan loop untuk menunggu sangat tidak direkomendasikan, alasannya buang – buang rsource CPU dan juga jika terjadi interrupt, loop akan terus berjalan tanpa henti
* Java sudah menyediakan solusi yang lebih baik dengan menambahkan method wait dan notify di java.lang.Object
* Artinya kitab isa membuat object apapun menjadi lock, dan gunakan wait() untuk menunggu, dan gunakan notify() untuk memberitahu bahwa data sudah tersedia
* Notify() akan memberi tahu thread lain yang sedang melakukan wait() bahwa proses bisa dilanjutkan



**Notify All**

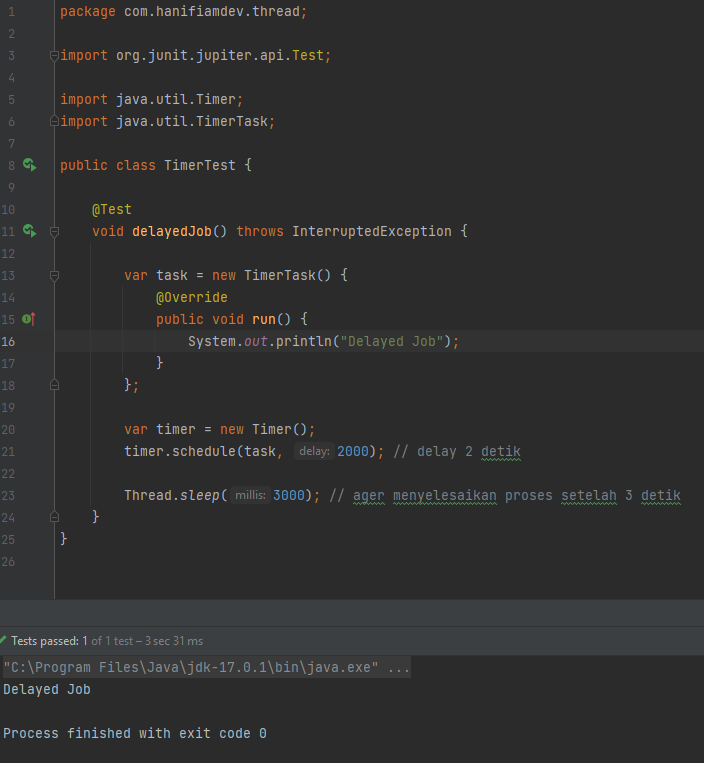
* Kadang ada kasus dimana sebuah lock ditunggu oleh banyak thread, notify() hanya memberi sinyal kepada suatu thread saja
* Jika kita ingin mengirim sinyal ke semua thread, thread bisa menggunakan method notifyAll()



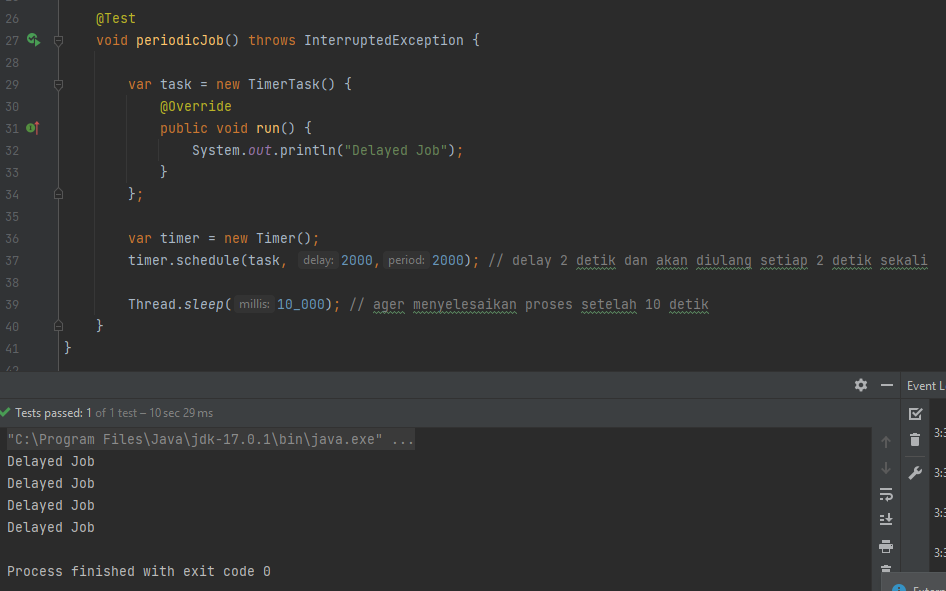
## **Timer**

* Timer merupakan class untuk memfasilitasi eksekusi job secara asynchronous di masa depan
* Timer bisa di schedule untuk berjalan satu kali ( delayed job ), atau bisa berjalan berulang kali ( repeated job )
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Timer.html>

**Timer Delayed Job**



**Timer Periodic Job**



## **High Level Concurrency Object**

**Concurrency Utilities**

* Pada versi Java 5, Java merilis fitur yang bernama Concurrency Utilities, fitur ini berisikan dukungan high level API untuk membuat aplikasi multithreaded lebih mudah
* Concurency Utilities memperkenalkan 3 packages baru di Java 5, yaitu
* java.util.concurrent
* java.util.concurrent.atomic, dan
* java.util.concurrent.locks

**Kenapa Butuh High Level API?**

* Membuat dan melakukan management Thread secara manual di java bukanlah hal bijak
* Mudah terjadi kesalahan, terutama seperti yang sudah kita jelaskan, masalah Race Condition dan Deadlock misalnya
* Selain itu, membuat aplikasi multithreaded menggunakan low level API ( menggunakan Thread langsung ) tidak terlalu produktif dan flexible
* Oleh karena itu, dalam pembuatan aplikasi sehari – hari, biasanya kita akan jarang sekali melakukan management thread secara manual, biasanya kita akan gunakan Concurrency Utilities

## **Threadpool**

**Management Thread itu Sulit**

* Thread merupakan object yang sangat mahal, pembuatan object thread bisa memakan memory 512kb-1mb
* Jika kita tidak mengatur pembuatan thread, bisa jadi aplikasi kita akan cepat kehabisan memory sehingga bisa menyebabkan error OutOfMemory
* Selain itu, membut object thread baru tidak semurah membuat object class biasa, oleh karena itu sebenarnya pembuatan thread secara manual tidaklah disarankan
* Di versi java modern ini, lebih disarankan menggunakan Threadpool untuk management thread

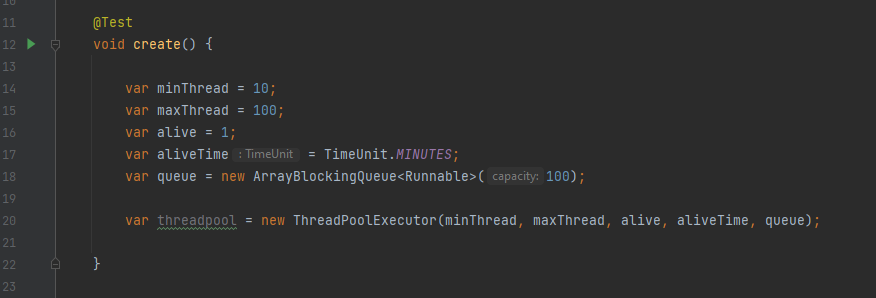
**Threadpool**

* Threadpool merupakan class yang digunakan untuk management thread
* Dengna threadpool, kita tidak perlu membuat thread secara manual, karena semua sudah diatur oleh threadpool
* Selain itu threadpool bisa melakukan reusable thread yang sudah selesai bekerja
* Threadpool di Java direpresentasikan dalam class bernama ThreadPoolExecutor

**Pengaturan Threadpool**

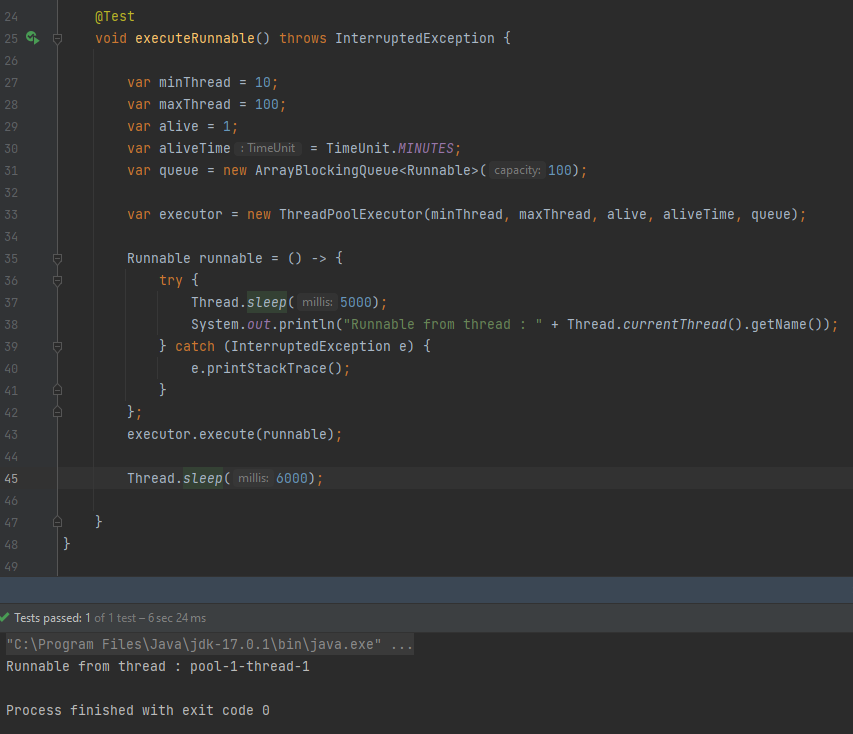
Ada beberapa yang harus kita atur ketika membuat threadpool

* core pool, minimal thread yang akan dibuat ketika threadpool dibuat
* max pool, maximal thread yang akan dibuat
* keep alive time, berapa lama thread akan dihapus jika tidak bekerja
* queue, antrian untuk menampung pekerjaan yang dikirm ke threadpool
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/ThreadPoolExecutor.html>



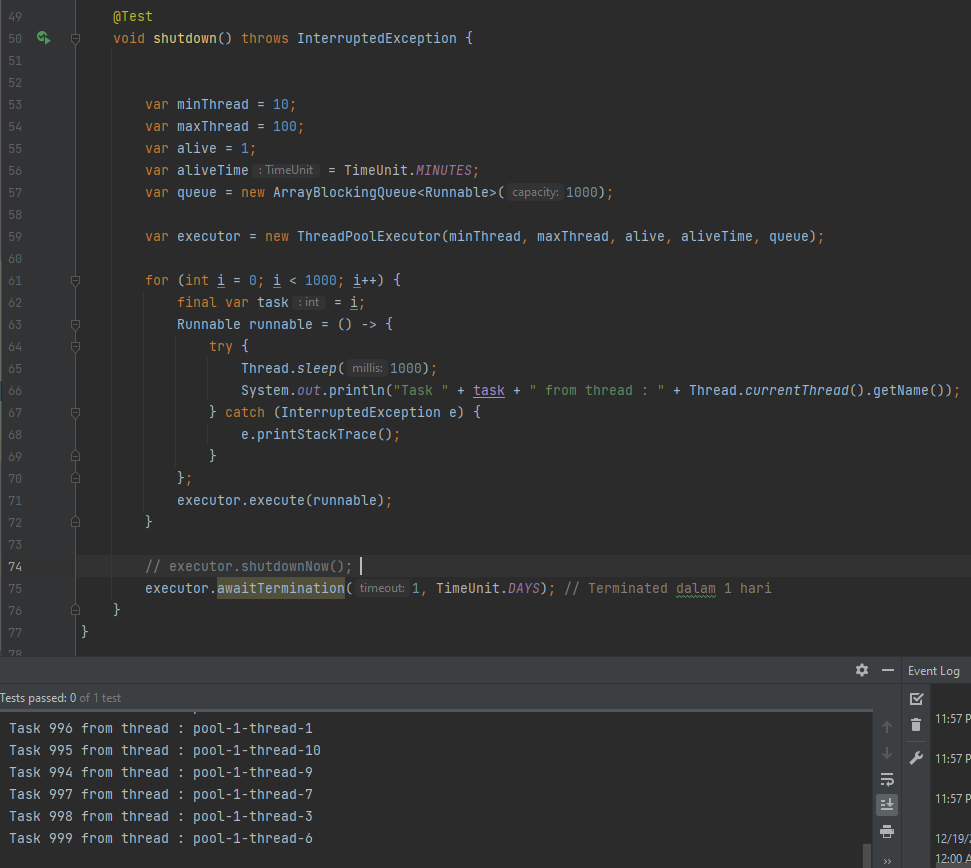
**Eksekusi Runnable**

* Untuk melakukan eksekusi Runnable, kitab isa menggunakan method execute() milik threadpool
* Secara otomatis data runnable akan dikirim ke queue threadpool untuk dieksekusi oleh thread yang terdapat di threadpool



**Menghentikan Threadpool**

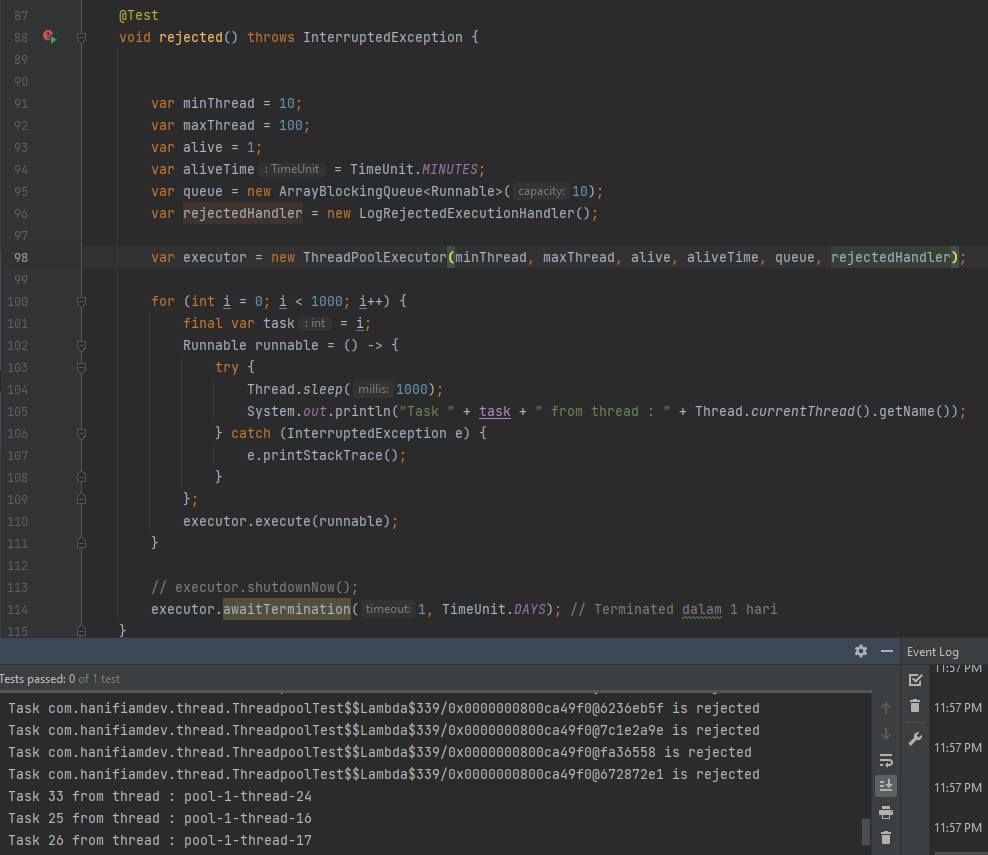
* Jika kita sudah selesai menggunakan threadpool, dan tidak akan menggunakannya lagi, ada baiknya kita hentikan dan matikan ThreadPool nya
* Caranya kita bisa menggunakan method shutdown() untuk menghentikan threadpool, jika ada pekerjaan yang belum dikerjakan, maka akan di ignore
* Atau gunakan shutdownNow() untuk menghentikan threadpool, namun pekerjaan yang belum dikerjakan akan dikembalikan
* Atau jika kita ingin menunggu sampai threadpool selesai, kitab isa gunakan awaitTermintaion()



**Rejected Handler**

* Apa yang terjadi jika queue penuh dan thread juga semua sedang bekerja?
* Maka secara otomatis di handle oleh object RejectedExecutionHandler
* Secara default, implementasi rejected handler akan mengambalikan exception RejectedExecution ketika kita submit(Runnable) pada kondisi queue penuh dan thread sedang bekerja semua
* Jika kita ingin mengubahnya, kita bisa membuat RejectedExecutionHandler sendiri

Berikut buat simulasi dimana queuenya dibuat 10 antrian supaya jika antrian penuh dan kemudian akan membuat thread baru sampai maksimal threadnya penuh yang akhirnya karena semua thread sibuk akan mentrigger Kelas RejectedExecutionHandler



## **ExecutorService**

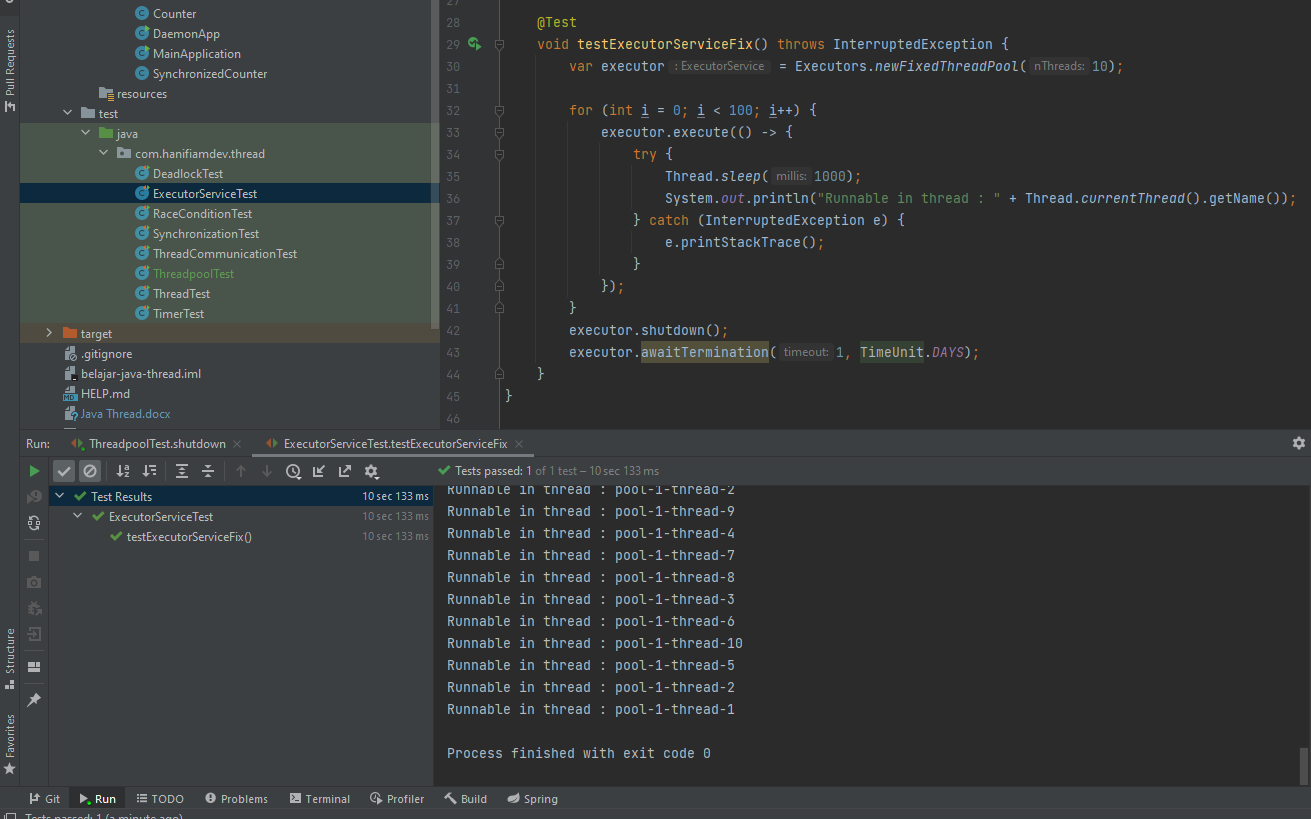
* Jika kita perhatikan, ThreadPoolExecuter merupakan implementasi dari interface Executor dan ExecutorService
* Jadi saat kita membuat ThreadPoolExecutor sebenarnya kita membuat Executor Service
* Dan sebenarnya pembuatan Threadpool secara manual jarang dilakukan, kecuali pada kasus kita benar-benar butuh melakukan tuning Threadpool
* Namun dalam kebanyakan kasus, kita jarang sekali membuat Threadpool secara manual
* Rata-rata, biasanya untuk mengeksekusi Runnable, biasanya kita akan menggunakan Executor Service
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/ExecutorService.html>

**Executors**

* Karena ExecutorService adalah interface, jadi pembuatan object ExecutorService salah satu nya adalah menggunakan ThreadPoolExecutor
* Namun ada yang lebih mudah, yaitu menggunakan class Executors
* Executor merupakan class utility untuk membantu kita membuat object ExecutorService secara mudah
* Sebenarnya implementasi Executors pun menggunakan ThreadPoolExecutor, hanya saja kita tidak perlu terlalu pusing melakukan pengaturan threadpool secara manual
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/Executors.html>

**Executors Method**

|  |  |
| --- | --- |
| **Executors Method** | **Keterangan** |
| newFixedThreadPool(n) | Membuat threadpool dengan jumlah pool min dan max fix |
| newSingleThreadExecutor() | Membuat threadpool dengan jumlah pool min dan max 1 |
| newCacheThreadPool() | Membuat threadpool dengan jumlah thread bisa bertambah tidak terhingga |

newFixedThreadPool(n), ketika n bernilai 10,,maka proses akan berlangsung 10 thread secara bergantian

## **Future**

**Callable<T>**

* Sebelumnya kita selalu menggunakan Runnable untuk mengirim perintah ke thread, namun pada Runnable, setelah pekerjaan selesai, tidak ada yang dikembalikan sama sekali, karena methodnya return void
* Callable mirip dengan Runnable, namun Callable mengembalikan data
* Selain itu Callable merupakan generic type, sehingga kita bisa tentukan tipe return data nya

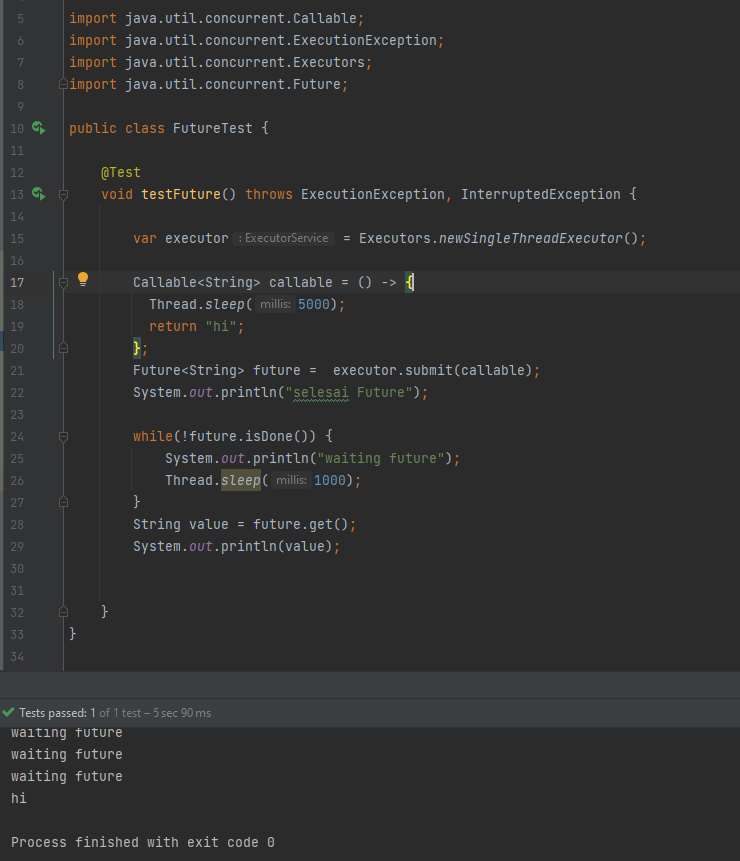
**Future<T>**

* Jika kita ingin mengeksekusi callable, kita bisa menggunakan method submit(Callable) pada ExecutorService, method submit(Callable) tersebut akan mengembalikan Future<T>
* Future merupakan representasi data yang akan dikembalikan oleh proses asynchronous
* Menggunakan Future, kita bisa mengecek apakah pekerjaan Callable sudah selesai atau belum, dan juga mendapatkan data hasil dari Callable

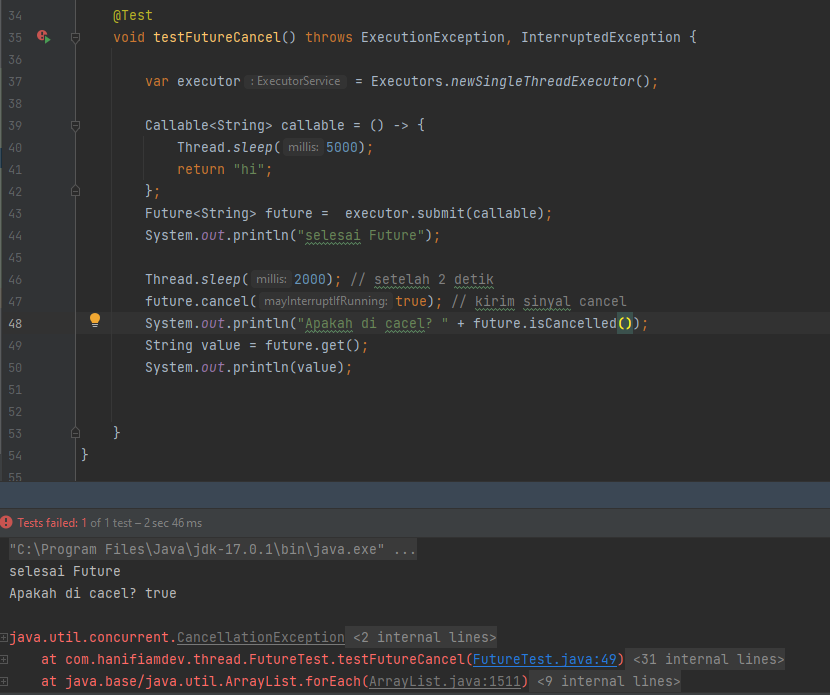
**Future Method**

|  |  |
| --- | --- |
| Future Method | Keterangan |
| T get() | Mengambil result data, jika belum ada, maka akan menunggu sampai ada |
| T get(timeout, time unit) | Mengambil result data, jika belum ada, maka akan menunggu sampai timeout |
| void cancel(mayInterrupt) | Membatalkan proses callable, dan apakah diperbolehkan di interrupt jika sudah terlanjur berjalan |
| boolean isCancelled() | Mengecek apakah future dibatalkan |
| boolean isDone() | Mengecek apakah future telah selesai |

**Membuat Future**

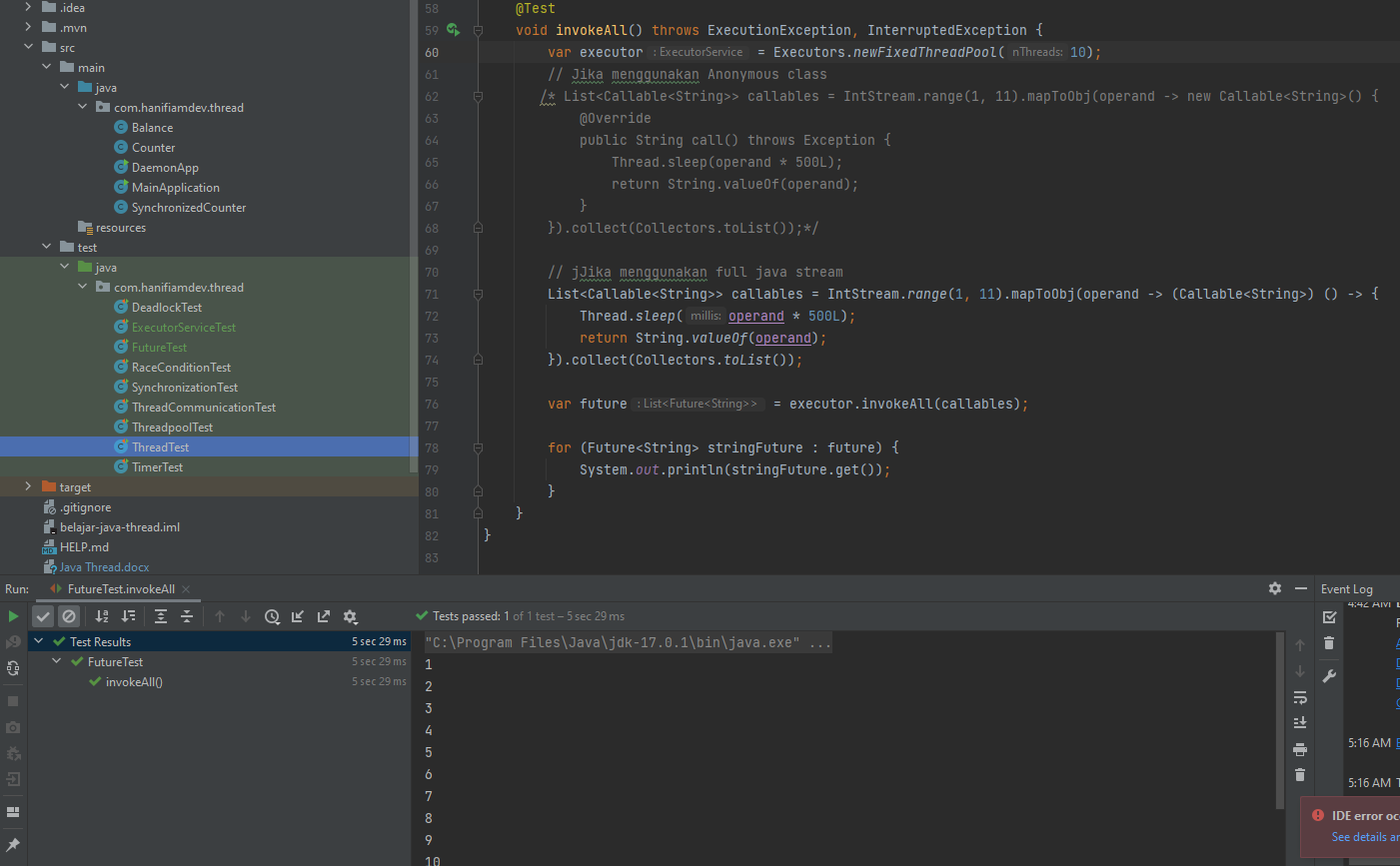


**Membatalkkan Future**



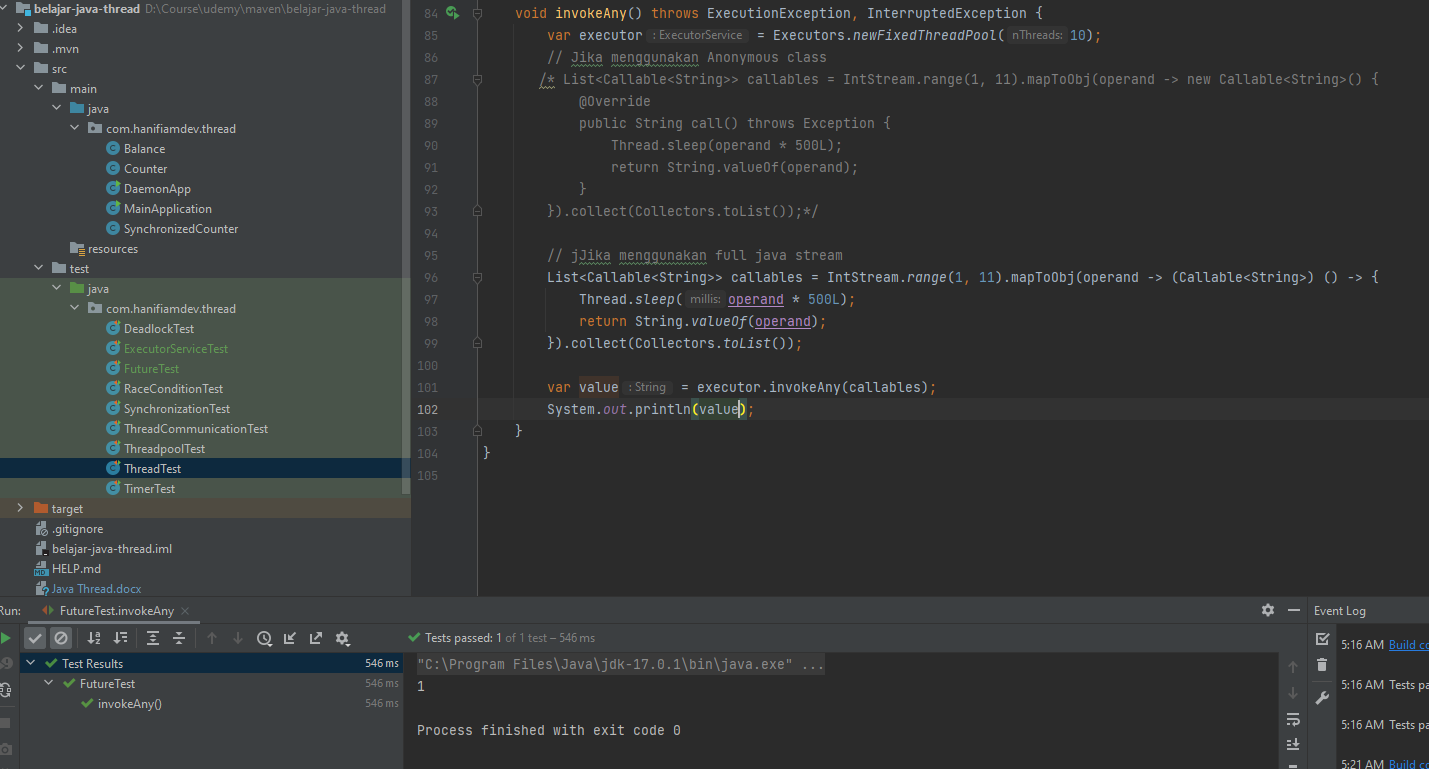
**Invoke All**

* ExecutorService memiliki method bernama invokeAll(Collection<Callable<T>>) untuk mengeksekusi banyak Callable secara sekaligus
* Ini cocok ketika ada kasus kita ingin menjalankan proses asynchronous secara parallel sebanyak jumlah thread di threadpool
* Hal ini bisa mempercepat proses dibanding kita eksekusi satu persatu



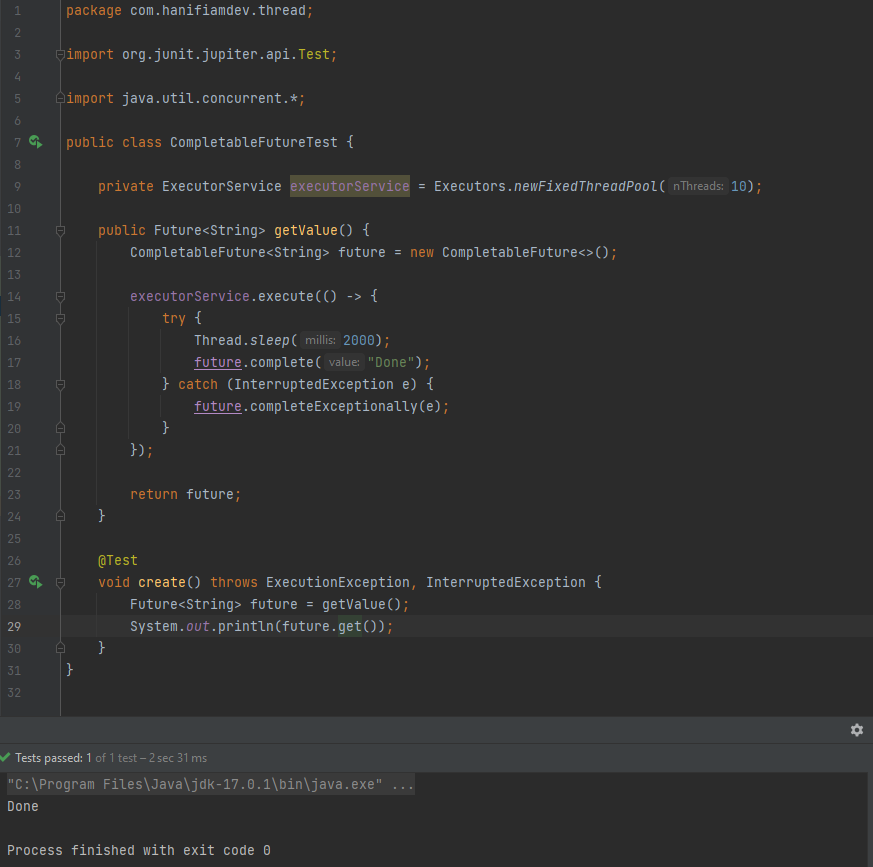
**Invoke Any**

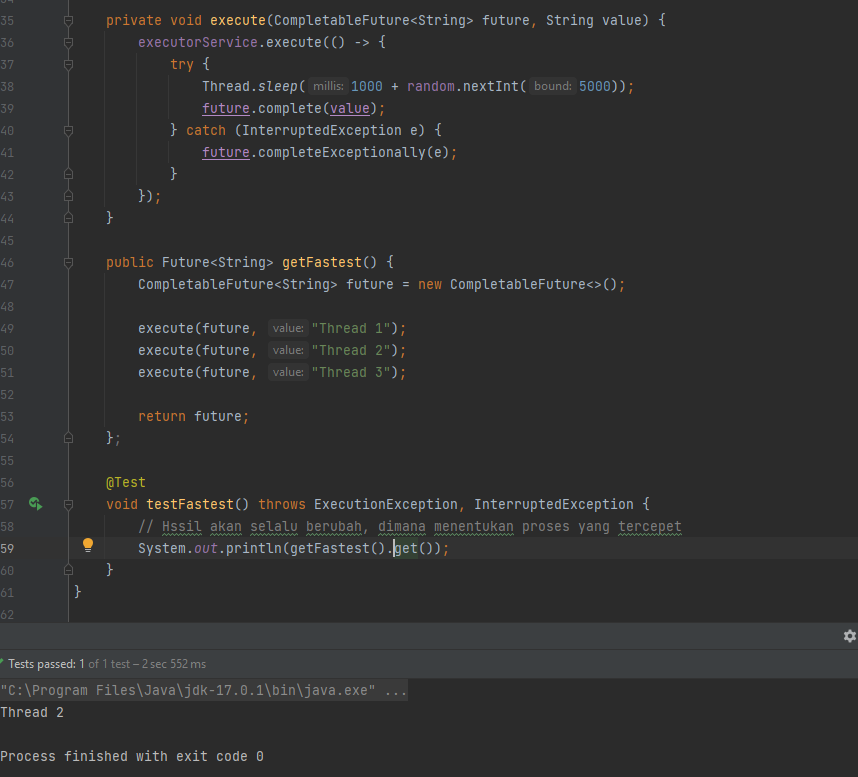
* Kadang ada kasus dimana kita ingin mengeksekusi beberapa proses secara asynchronous, namun ingin mendapatkan hasil yang paling cepat
* Hal ini bisa dilakukan dengan menggunakan method invokeAny(Collection<Callable<T>>)
* invokeAny() akan mengembalikan result data dari Callable yang paling cepat mengembalikan result



## **Complateable Future**

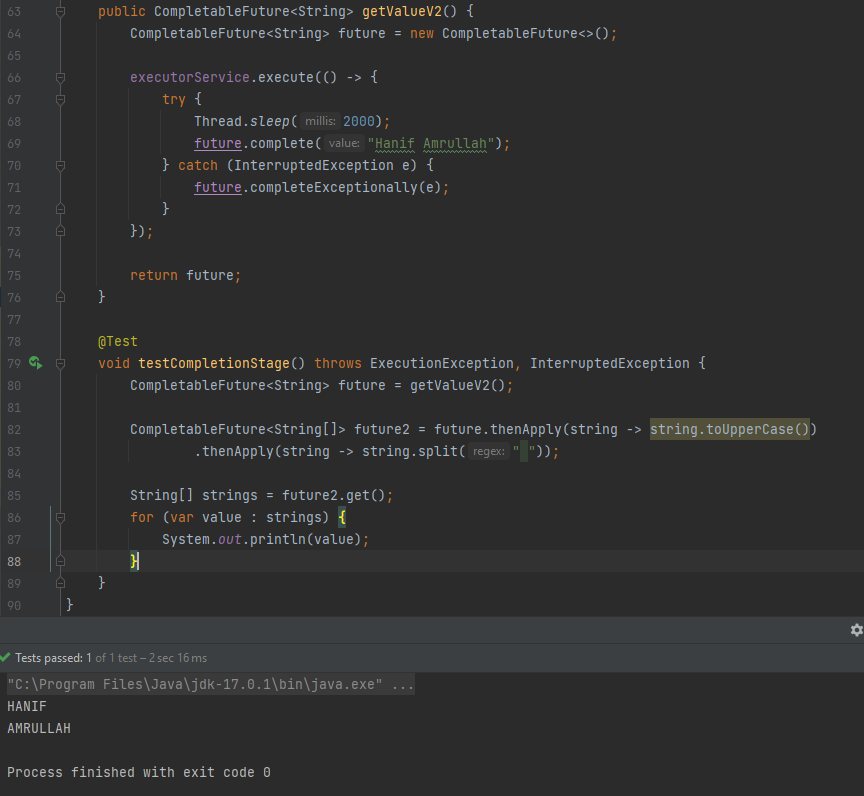
* Pada Java 8, terdapat sebuah class baru bernama CompletableFuture, ini merupakan implementasi Future yang bisa kita set datanya secara manual
* CompleteableFuture sangat cocok ketika kita misal perlu membuat future secara manual, sehingga kita tidak memerlukan Callable
* Untuk memberi value terhadap CompleteableFuture secara manual, kitab isa menggunakan method complete(value) atau completeExceptionally(error) untuk error
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/CompletableFuture.html>





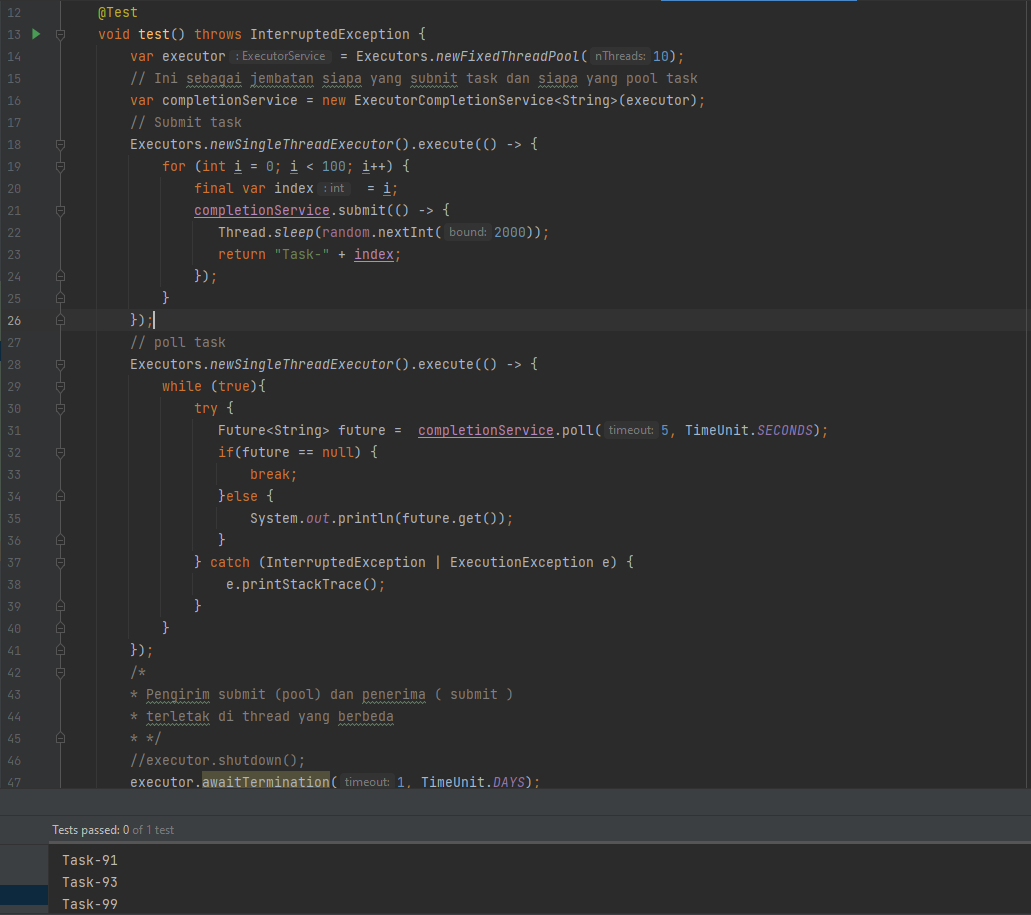
**CompletionStage**

* CompletableFuture merupakan turunan dari interface CompletionStage
* CompletionStage merupakan fitur dimana kita bisa menambahkan asynchronous computation,tanpa harus menunggu dulu data dari Future nya ada
* CompletionStage sangat mirip dengan operation di Java Stream, hanya saja tidak sekomplit di Java Stream
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/CompletionStage.html>



## **Completion Service**

* CompletionService merupakan sebuah interface yang digunakan untuk memisahkan antara bagian yang mengeksekusi asynchronous task dan menerima hasil dari task yang sudah selesai
* Kadang ada kebutuhan misal kita butuh mengejakan sesuatu secara parallel, lalu ada satu thread yang melakukan eksekusi task dan satu thread menunggu hasil nya
* Kita bisa menggunakan CompletionService untuk menunggu itu
* Implementasi interface CompletionService adalah class ExecutorCompletionService
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/CompletionService.html>
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/ExecutorCompletionService.html>



## **Scheduled Executor Service**

* ExecutorService memiliki sub child interface bernama ScheduledExecutorService
* Fitur tambahan di ScheduledExecutorService adalah, kita bisa melakukan asynchronous task yang terjadwal
* Hal ini sangat cocok untuk kasus delayed job (pekerjaan yang butuh ditangguhkan pengerjaan) dan periodic job
* ScheduledExecutorService merupakan fitur yang bisa menggantikan low level penggunaan Timer
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/ScheduledExecutorService.html>

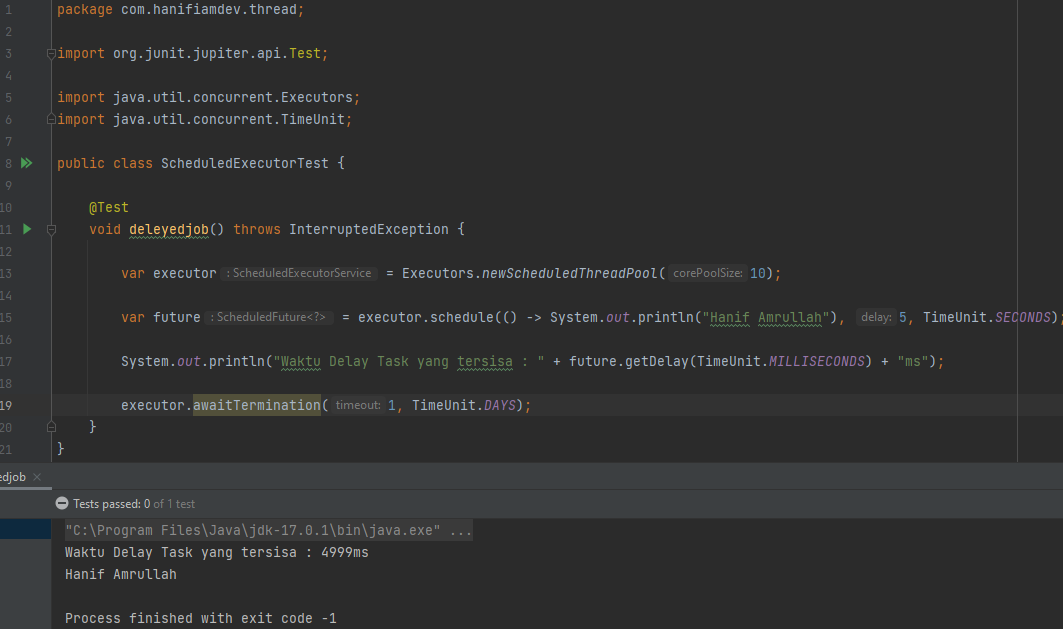
**ScheduledFuture**

* Hampir semua method di ScheduledExecutorService mengembalikan data ScheduledFuture
* ScheduledFuture sebenarnya mirip dengan Future, yang membedakan, dia adalah turunan dari interface Delayed, yang memiliki method untuk mendapatkan informasi waktu delay
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/ScheduledFuture.html>

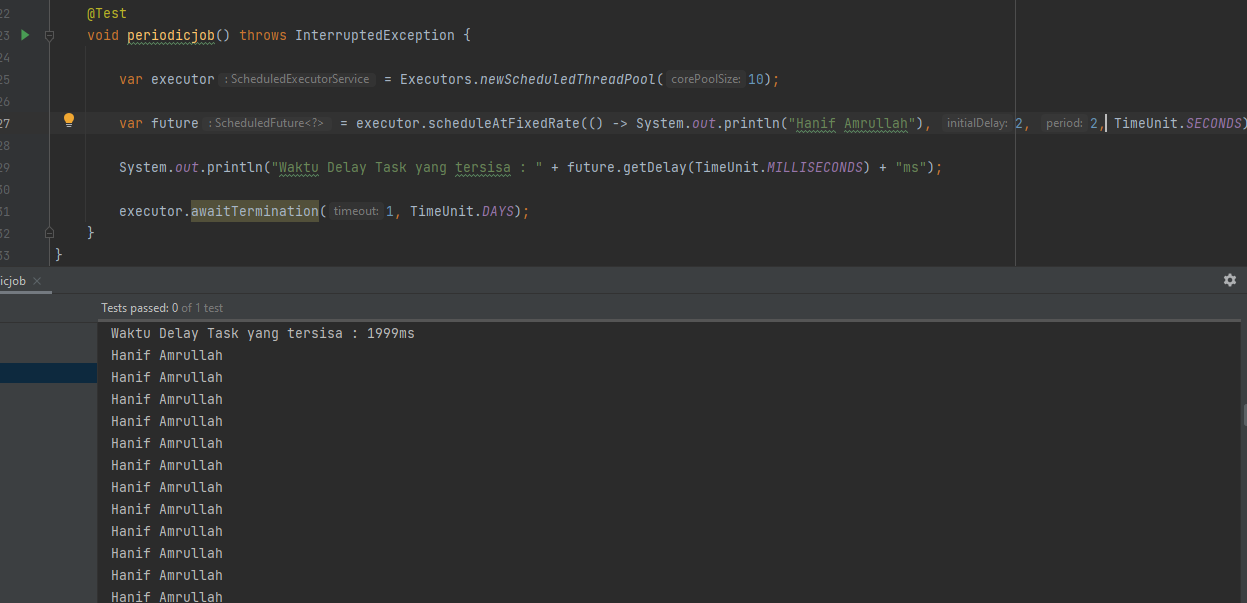
**Membuat ScheduledExecutorService**

* Untuk membuat ScheduledExecutorService kitab isa menggunakan implementasi class ScheduledThreadPoolExecutor
* Atau jika ingin mudah, kitab isa gunakan class Executors, terdapat method newSingleThreadScheduledExecutor() dan newScheduledThreadPool(poolSize) untuk membuat ScheduledExecutorService
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/ScheduledThreadPoolExecutor.html>

**Delayed Job**



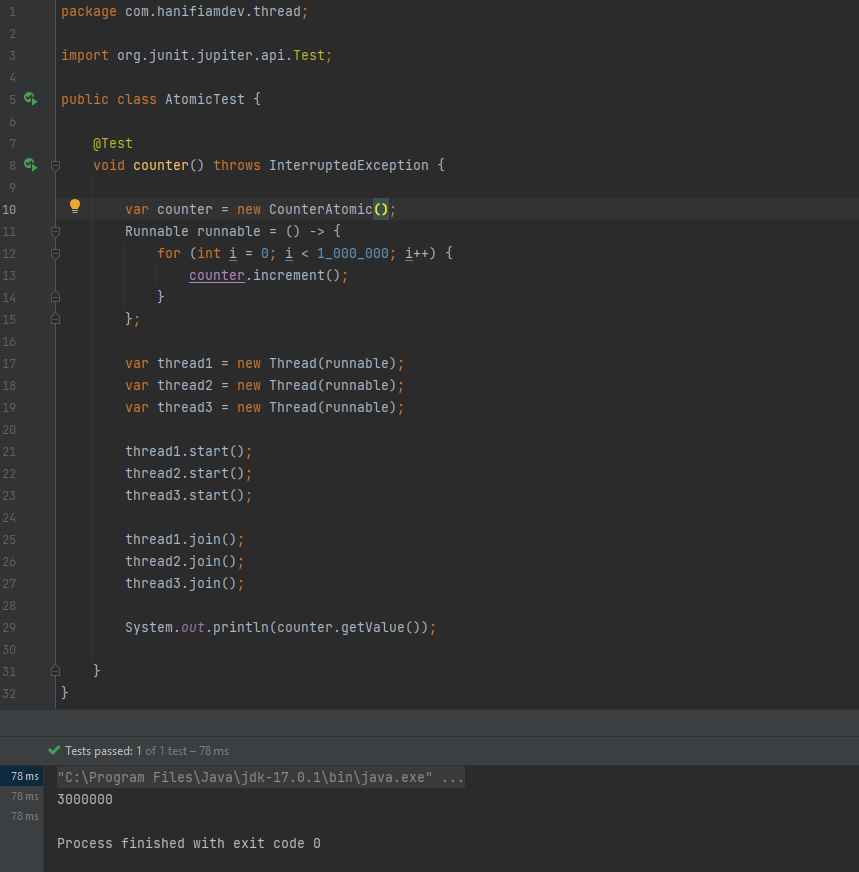
**Periodic Job**



## **Atomic**

**Atomic Package**

* Java menawarkan sebenarnya package atomic yang berisikan class-class yang mendukung lock-free dan thread-safe programming pada single variable
* Setiap object Atomic class akan mengelola data yang diakses dan diupdate menggunakan method yang telah disediakan
* Atomic class melakukan implementasi Compare-and-Swap untuk mendukung synchronization
* Dengan menggunakan Atomic, kita perlu lagi menggunakan synchronized secara manual
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/atomic/package-summary.html>
* <https://en.wikipedia.org/wiki/Compare-and-swap>



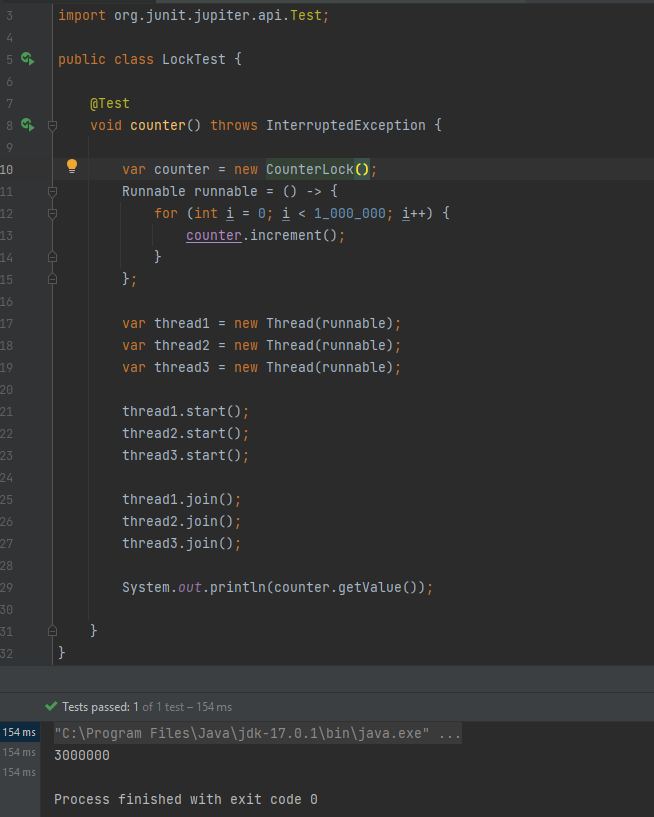
## **Lock**

**Locks Package**

* Java menyediakan high level concurrency package untuk melakukan locking atau waiting for conditions
* Ini adalah alternatif dari low level synchronized dan manual wait dan notify
* Untuk saat ini, sangat disarankan menggunkan package locks dibandingkan menggunakan low level sebelumnya yang sudah kita bahas
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/locks/package-summary.html>

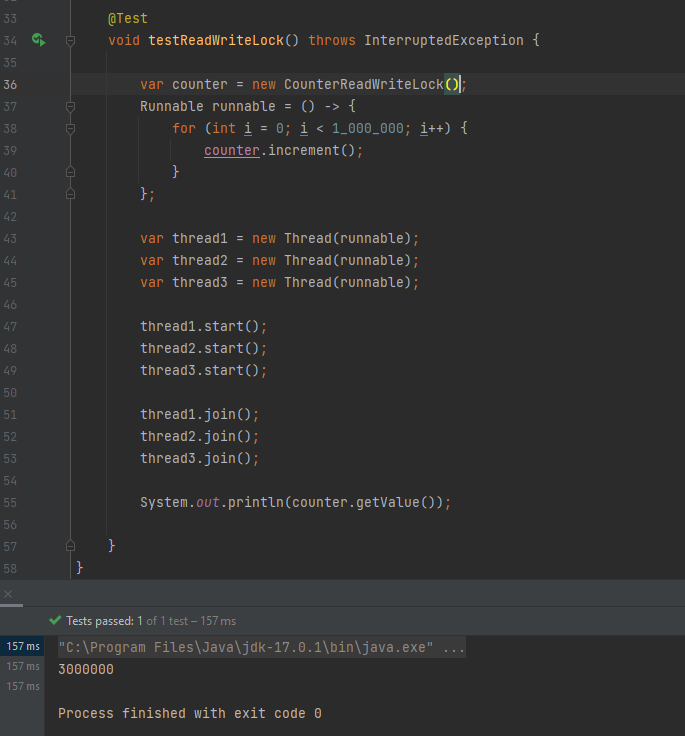
**Locks Interface**

* Lock interface merupakan alternatif implementasi dari synchronized method dan synchronized statement
* Untuk melakukan lock, kita bisa gunakan method lock() dan setelah selesai, kita bisa menggunakan method unlock() untuk melepaskan lock
* Implementasi dari interface Lock adalah class ReentrantLock
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/locks/Lock.html>



**ReadWriteLock**

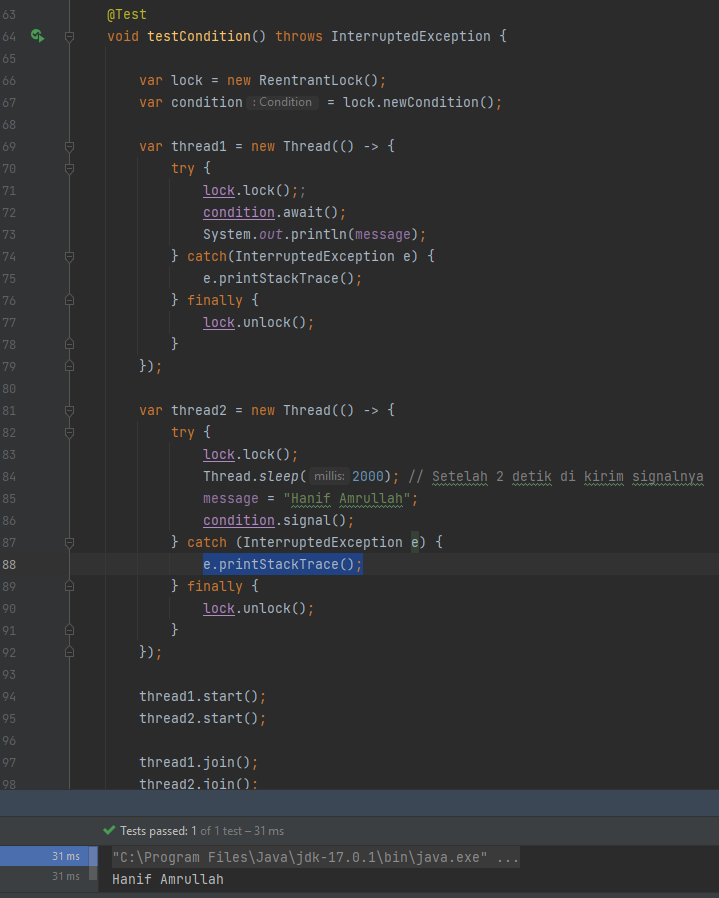
* Kadang ada kondisi dimana kita ingin membedakan lock antara operasi update dan operasi get
* Untuk kasus seperti ini, kita bisa membuat dua buah variable Lock
* Namun, di Java disediakan cara yang lebih mudah, yaitu menggunakan interface ReadWriteLock
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/locks/ReadWriteLock.html>



**Condition Interface**

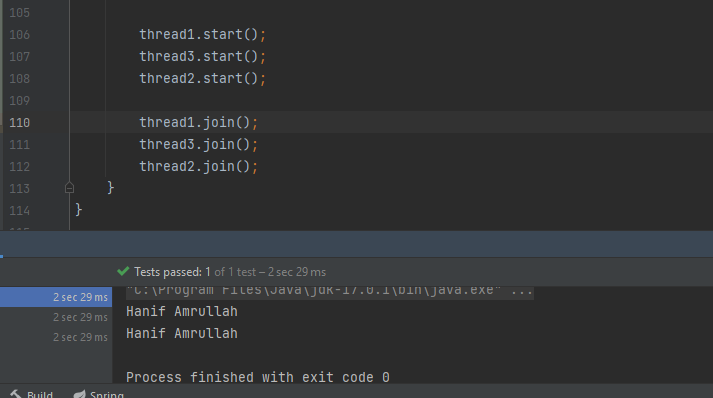
* Condition merupakan alternatif lain dari monitor method(wait, notify, dan notifyAll)
* Pada Java modern saat ini, sangat disarankan menggunakan Condition dibanding monitor method
* Condition memiliki method wait() untuk menunggu, signal() untuk mentrigger satu thread, dan signalAll() untuk mentrigger semua thread yang menunggu
* Cara pembuatan Condition, kitab isa menggunakan method newCondition() milik Lock
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/locks/Condition.html>

signal()



signalAll()





## **Synchronizer**

* Pada package concurrent, terdapat banyak sekali class untuk melakukan synchronizer, ini sebenarnya improvement dari locks, namun digunakan pada kasus-kasus tertentu
* Isi dari class-class Synchronizer banyak menggunakan locks, namun kita tidak perlu melakukannya secara manual, karena sudah diatur secara otomatis oleh class-class nya sendiri

**Jenis Synchronizer**

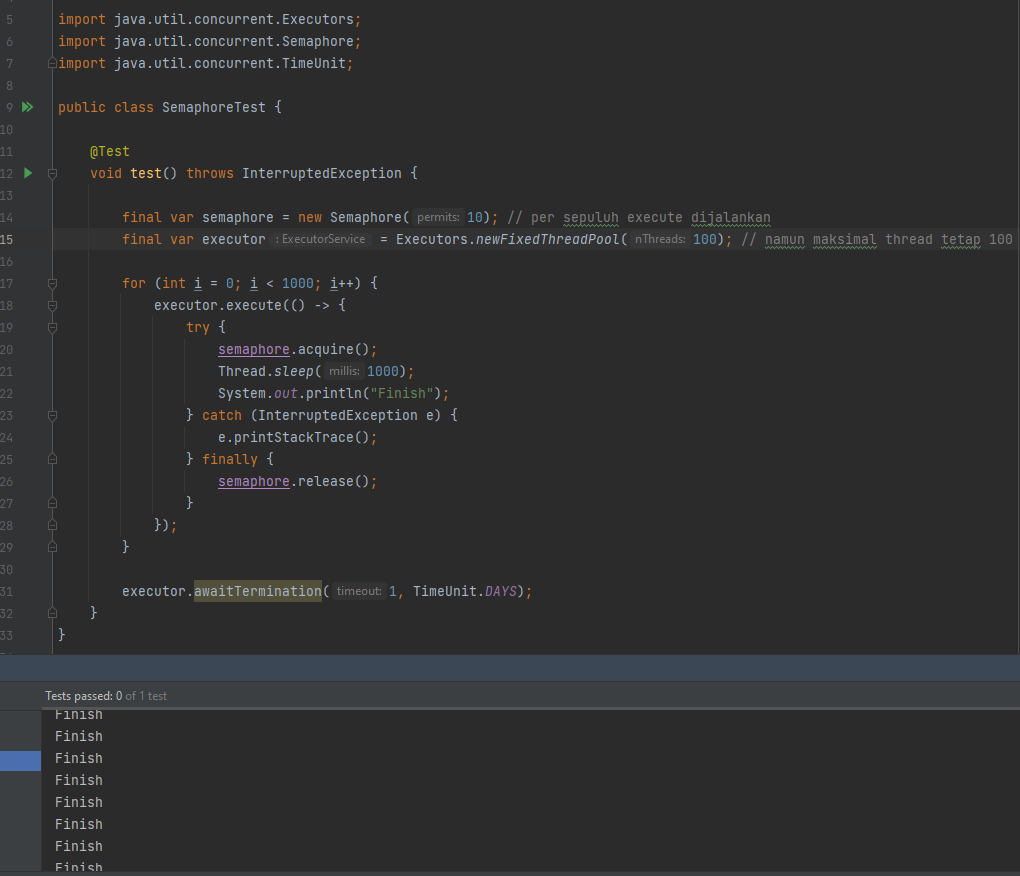
Ada banyak sekali class – class yang bis akita gunakan untuk synchronizer, seperti

* Semaphore
* CountDownLatch
* CyclicBarrier
* Phaser, dan
* Exchanger

Semua kita akan coba bahas satu persatu

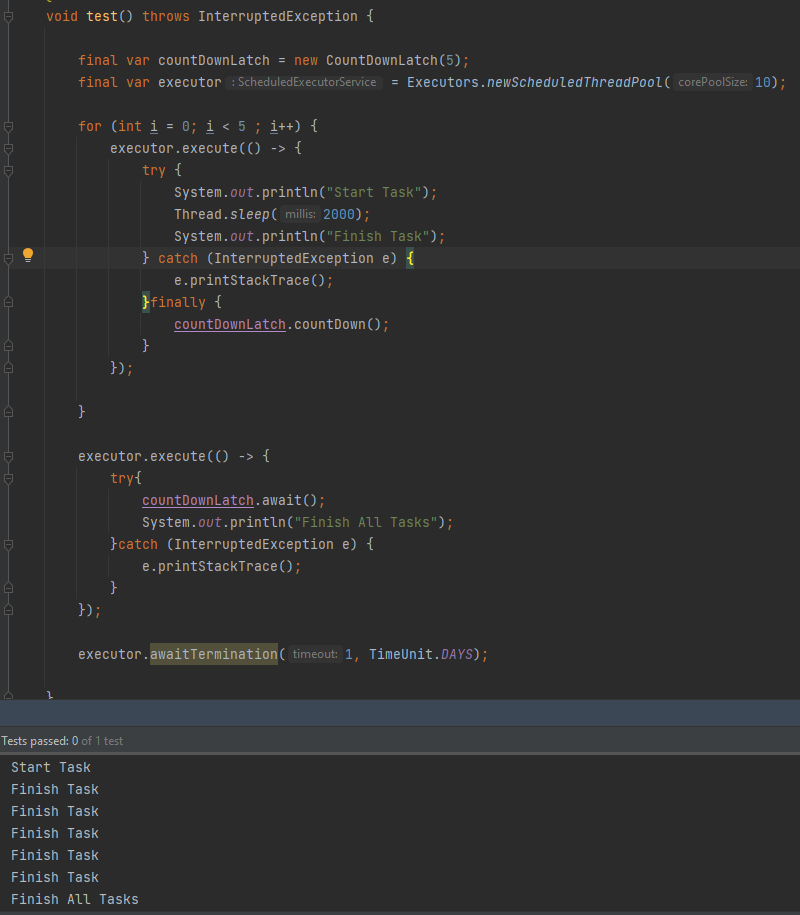
## **Semaphore**

* Semaphore merupakan class yang digunakan untuk manage data counter
* Nilai counter bisa naik, namun ada batas maksimal nya, jika batas maksimal nya udah tercapai, semua thread yang akan mencoba menaikkan harus menunggu, sampai ada thread lain yang menurunkan nilai counter
* Semaphore cocok sekali misal untuk menjaga agar thread berjalan pada maksimal total counter yang sudah kita tentukan
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/Semaphore.html>



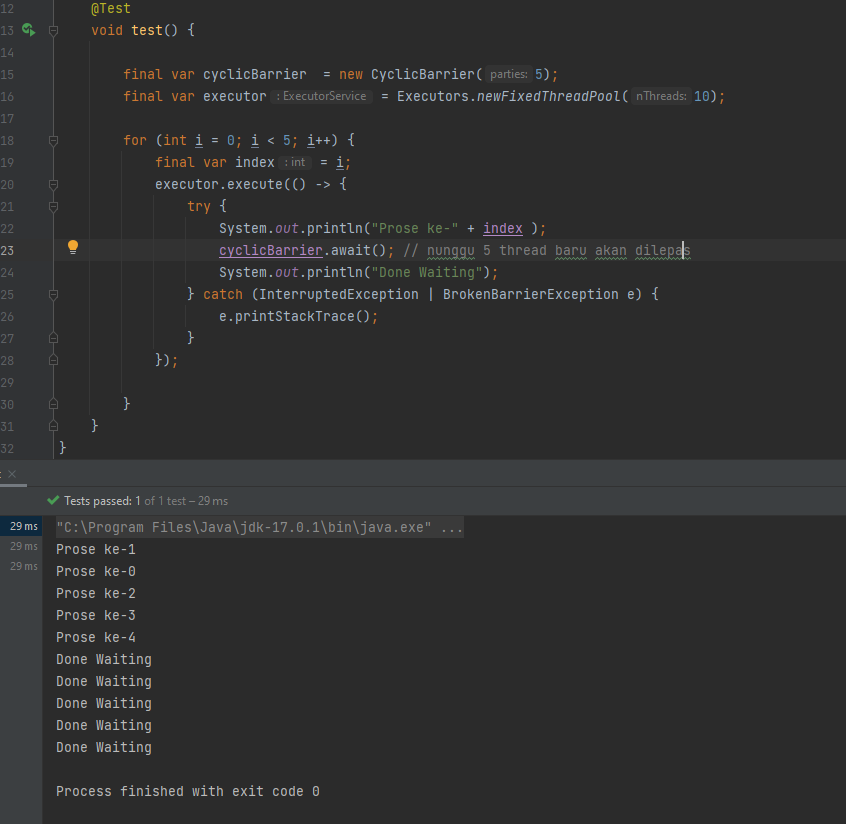
## **Count Down Latch**

* CountDownLatch merupakan synchronizer yang digunakan untuk menunggu beberapa proses selesai, cara kerjanya mirip dengan Semaphore, yang membedakan adalah pada CountDownLatch, counter diawal sudah ditentukan
* Setelah proses selesai, kita akan menurunkan counter
* Jika counter sudah bernilai 0, maka melakukan wait bisa lanjut berjalan
* CountDownLatch cocok jika kita misal ingin menunggu beberapa proses yang berjalan secara asynchronous sampai semua proses selesai
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/CountDownLatch.html>



## **Cyclic Barrier**

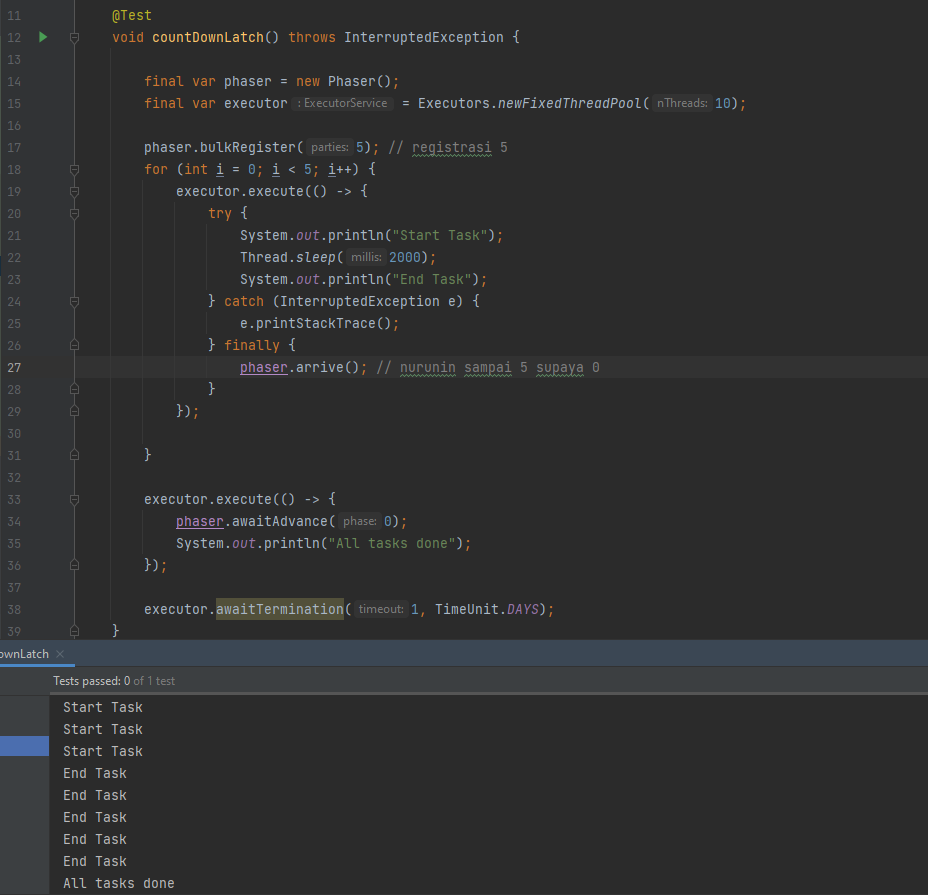
* CyclicBarrier merupakan fitur yang bis akita gunakan untuk saling menuggu, sampai jumlah thread yang menunggu terpenuhi
* Diawal kita akan tentukan berapa jumlah thread yang menunggu, jika sudah terpenuhi, maka secara otomatis proses menunggu akan selesai
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/CyclicBarrier.html>



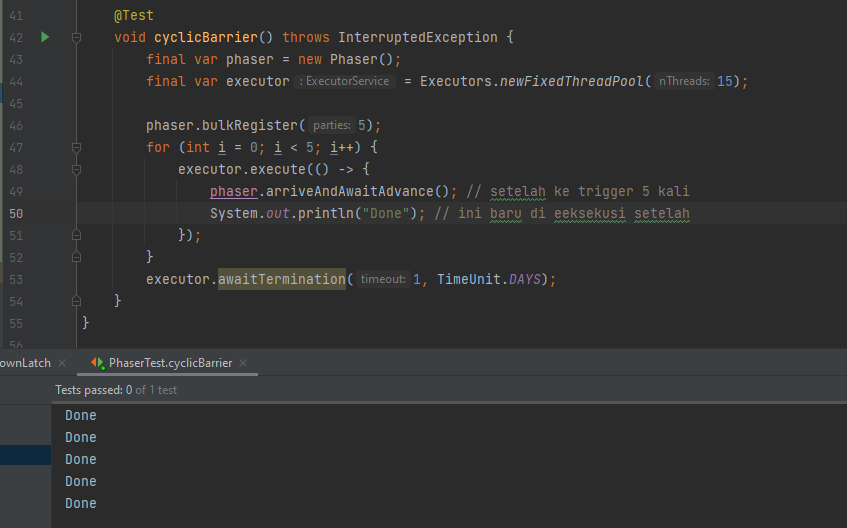
## **Phaser**

* Phaser merupakan fitur synchronizer yang mirip denganm CyclicBarrier dan CountDownLatch namun lebih flexible
* Sebelumnya, untuk jumlah counter atau thread nya sudah ditentukan di awal. Namun pada Phaser, bisa berubah dengan menggunakan method register() atau bulkRegister(int), dan untuk menurunkan bisa menggunakan method arrive…(), atau bisa menggunakan await…(int) untuk menunggu sampai jumlah yang register tertentu
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/Phaser.html>

**Phaser Sebagai CountDownLatch**

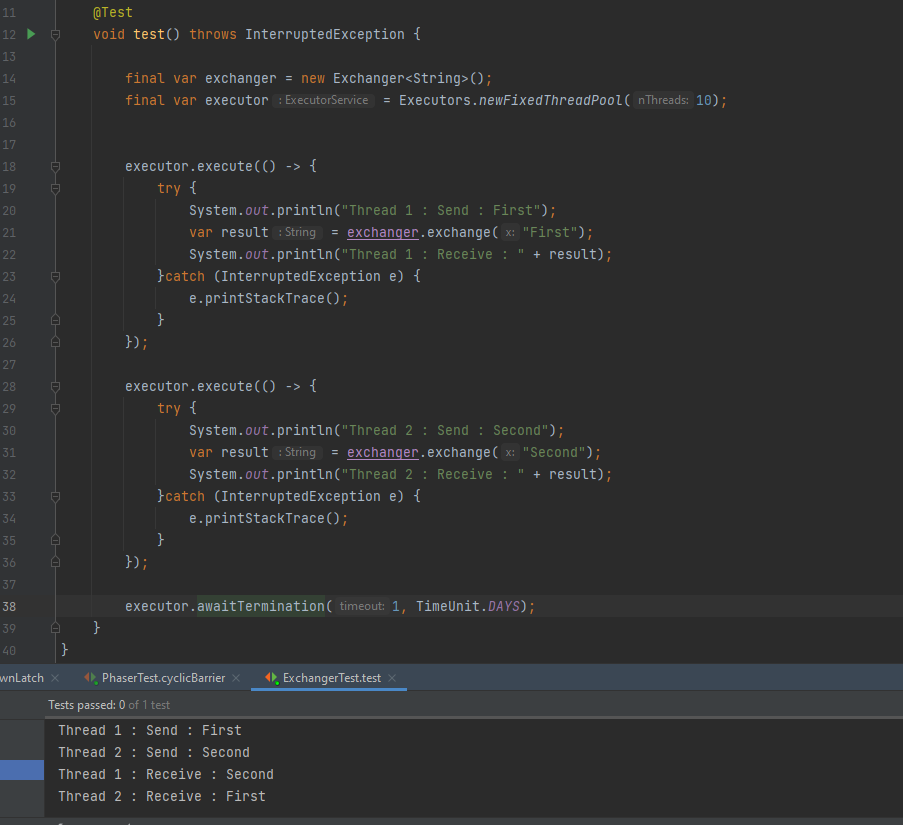


**Phaser Sebagai CyclicBarrier**



## **Exchager**

* Exchanger merupakan fitur synchronizer untuk melakukan pertukaran data antar thread
* Jika data belum tersedia, maka thread yang melakukan pertukaran akan menunggu sampai ada thread lain yang melakukan pertukaran data
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/Exchanger.html>



## **Concurrent Collection**

* Java menyediakan Collection yang banyak digunakan untuk proses concurrent programming
* Tidak seperti kebanyakan Collection di pakage java.util, Collection yang terdapat di java.util.concurent memang dikhususkan untuk concurrent programming, sehingga thread safe

**Jenis Concurent Collection**

Secara garis besar,concurrent collection di Java terbagi menjadi dua interface

* BlockingQueue, merupakan turunan dari Queue, dan dikhususkan untuk tipe collection FIFO (first in first out), seperti yang sebelumnya sudah kita bahas sekilas di ThreadPoool Queue
* ConcurrentMap, merupakan turunan dari Map, dan dikhususkan untuk Mao yang thread safe dibanding implementasi Map di Java Collection

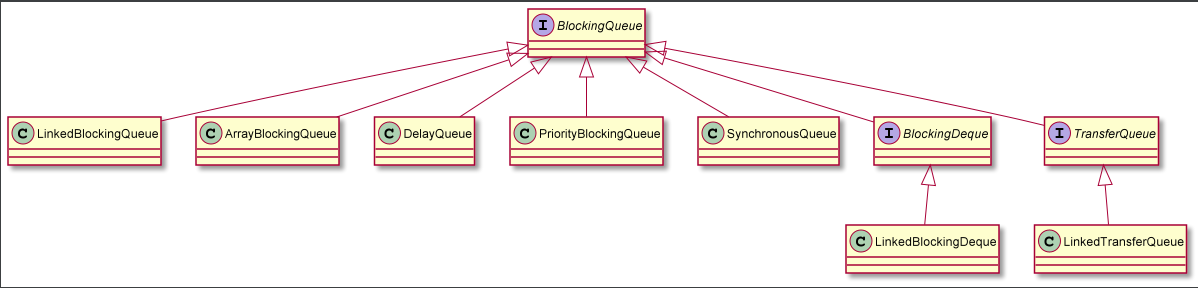
Kita akan coba bahas satu-satu chapter selanjutnya

## **Blocking Queue**

* BlockingQueue merupakan turunan dari Queue, dari namanya kita tahu bahwa ini adalah collection dengan tipe antrian, yang membedakan sifat dari BlockingQueue adalah dia aman diakses oleh beberapa thread, baik itu yang memasukkan data, atau mengambil data
* BlockingQueue mendukung method wait ketika mengambil data, dan juga wait ketika menyimpan data
* Jika queue kosong, thread yang mengambil data akan diminta untuk menunggu sampai data ada
* Dan jika queue penuh, thread yang mengambil data akan diminta untuk menunggu sampai data ada tempat kosong
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/BlockingQueue.html>



**Class Diagram Blocking Queue**



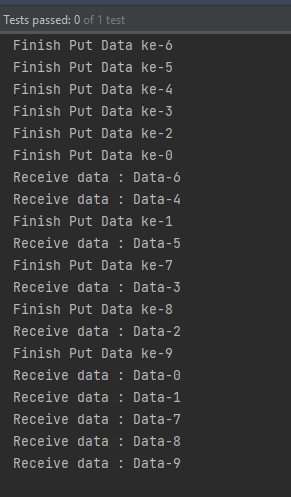
**Implementasi Blocking Queue**

* ArrayBlockingQueue, implementasi BlockingQueue dengan ukuran fix
* LinkedBlockingQueue, implementasi BlockingQueue dengan ukuran bisa berkembang
* PriorityBlockingQueue, implementasi BlockingQueue dengan otomatis berurut berdasarkan prioritas
* DelayQueue, implementasi BlockingQueue untuk tipe data Delayed, dimana data tidak bisa diambil sebelum waktu delay yang telah ditentukan
* SynchronousQueue, implementasi BlockingQueue dimana thread yang menambah data harus menunggu sampai ada thread yang mengambil data, begitu juga kebalikannya

**ArrayBlockingQueue**



Result ArrayBlockingQueue

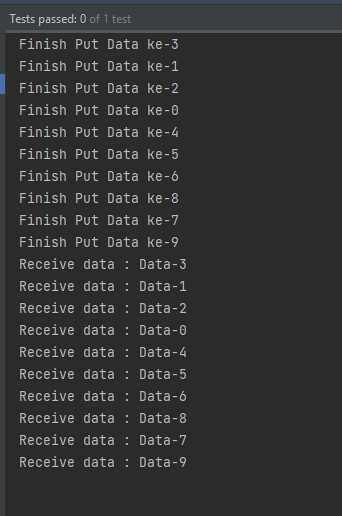


*Note : Jika 5 data sudah berhasil menyimpan data dan ada yang melakukan take maka data selanjutnya akan masuk begitu pula seterusnya*

**LinkedBlockingQueue**



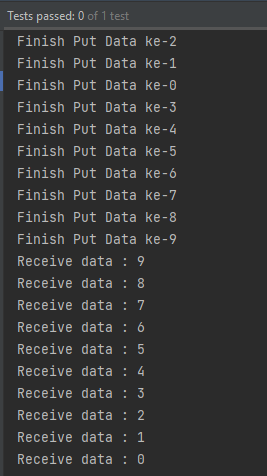
Result LinkedBlockingQueue



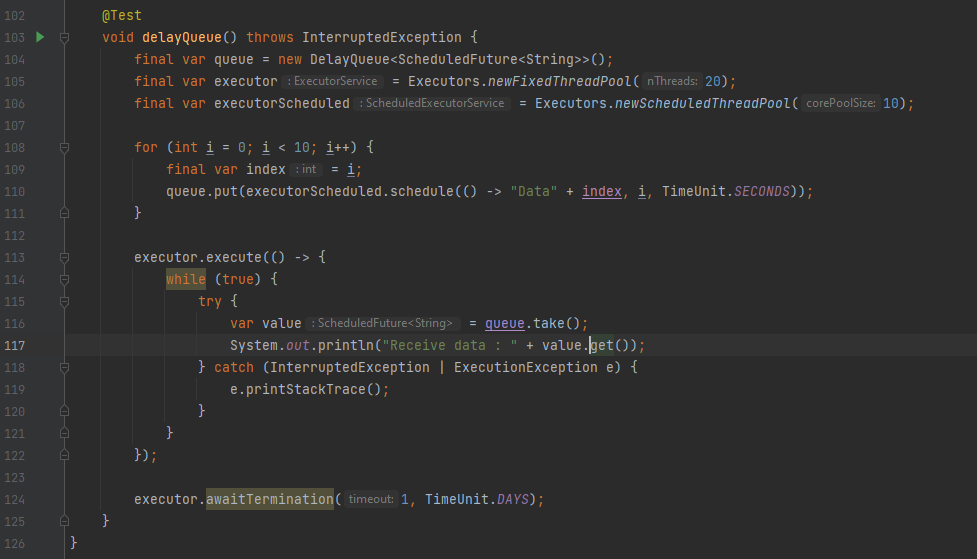
**PrioritiBlockingQueue**

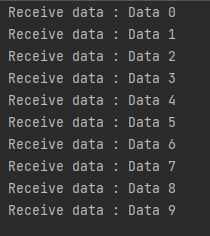


Result PrioritiBlockingQueue

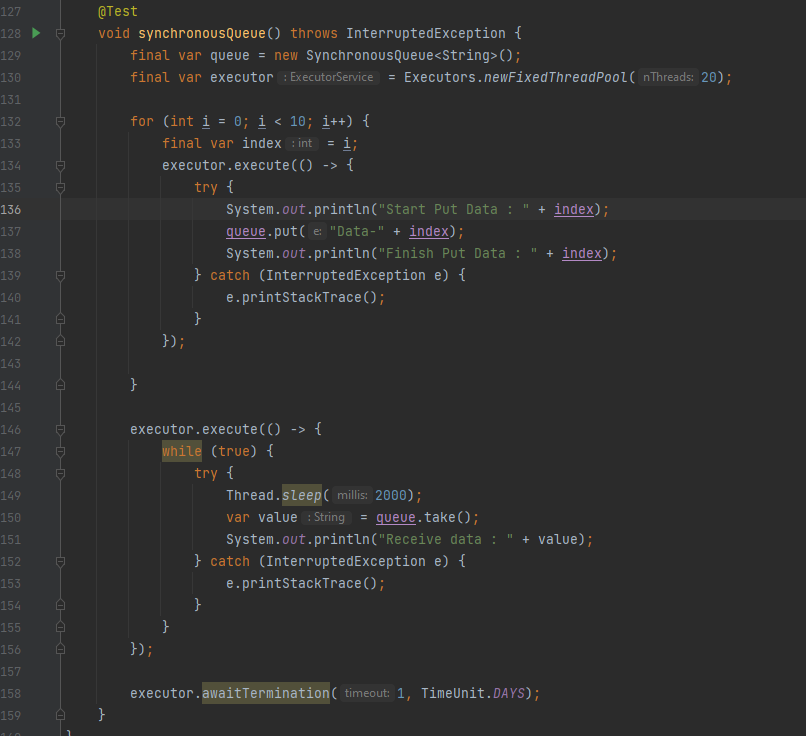


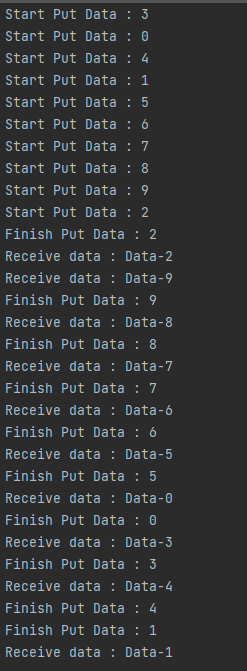
**DelayQueue**



Note : ketika “*queue.take()”* akan di tunggu sampai ada data , Data 0 akan keluar setelah 1 detik , Data 1 setelah 2 detik dst. Hal ini diakibatkan proses schedule “*executorScheduled.schedule(() -> "Data " + index, i, TimeUnit.SECONDS)*”

**SynchronousQueue**

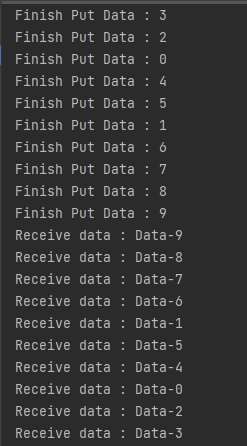


 Note : Melakukan put data selanjutnya ketika sudah melakukan take data sebelumnya

**BlockingDeque**

* BlockingDeque merupakan turunan dari BlockingQueue
* BlockingDeque tidak hanya mendukung FIFO(First in first out), tapi juga LIFO (last in last out)
* Implementasi BlockingDeque hanyala LinkedBlokingDeque
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/BlockingDeque.html>



Note : putLast sehingga menjadi index terakhir dan diambil memang index terakhir sehingga seperti tumpukan (LIFO). Karena memang BlockingDeque tidak hanya mendukung FIFO(First in first out), tapi juga LIFO (last in last out)

**TrransferDeque**

* TransferQueue Merupakan turunan BlockingQueue yang membolehkan pengirim data ke queue menunggu sampai data ada yang menerima
* Implementasi TransferQueue hanyalah LinkedTransferQueue
* <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/TransferQueue.html>



