Лекція 3. Типи пулів потоків.

Тема 1. Інтерфейс ExecutorService – основні методи для виконання та очікування завершення задач у пулі потоків.

Тема 2. Методи визначення стану потоку. Методи для зупинки потоків.

Тема 3. Типи пулів потоків – FixedThreadPool, CachedThreadPool, ScheduledThreadPool, SingleThreaded.

public class ThreadPoolExecutor extends AbstractExecutorService

ExecutorService, виконує надіслане завдання за допомогою одного або кількох об’єднаних потоків, зазвичай налаштованих за допомогою методів фабрики Executors.

Пули потоків вирішують дві різні проблеми: вони зазвичай забезпечують покращену продуктивність під час виконання великої кількості асинхронних завдань завдяки зменшенню накладних витрат на виклик кожного завдання, і вони забезпечують засоби обмеження та керування ресурсами. Кожен ThreadPoolExecutor також підтримує деяку базову статистику, наприклад кількість виконаних завдань.

ExecutorService клас надає багато настроюваних параметрів. Проте програмістам рекомендується використовувати

методи Executors.newCachedThreadPool() (необмежений пул потоків, з автоматичним відновленням потоків)

Executors.newFixedThreadPool(int) (пул потоків фіксованого розміру) і

Executors.newSingleThreadExecutor() (один фоновий потік), які попередньо налаштовують параметри для найбільш поширених сценарії використання.

Конструктори

ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue)

Цей конструктор створює новий ThreadPoolExecutor із заданими параметрами:

corePoolSize: Мінімальна кількість потоків в пулі. Екзек'ютор зберігає у пулі таку кількість потоків навіть якщо вони простоюють.

maximumPoolSize: Максимальна кількість потоків, які можуть бути активні в один і той же час.

unit: Часовий інтервал, який визначає одиницю виміру часу для параметра keepAliveTime.

workQueue: Черга, в яку задачі очікують на своє виконання, коли всі потоки зайняті.

ThreadFactory відповідає за створення нових потоків, якщо потрібно, а RejectedExecutionHandler визначає поведінку, коли задача не може бути прийнята до виконання (наприклад, коли черга повна і всі потоки зайняті).

Час підтримки

keepAliveTime: Час, протягом якого потоки, які перевищують кількість corePoolSize, можуть простоювати перш ніж будуть завершені.

Якщо пул наразі містить більше потоків, ніж corePoolSize, надлишкові потоки буде припинено, якщо вони були неактивні більше, ніж keepAliveTime (див. getKeepAliveTime(TimeUnit)).

Це забезпечує засіб для зменшення споживання ресурсів, коли пул не використовується активно. Якщо пізніше пул стане більш активним, будуть створені нові потоки. Цей параметр також можна динамічно змінювати за допомогою методу setKeepAliveTime(long, TimeUnit).

Використання значення Long.MAX_VALUE TimeUnit.NANOSECONDS фактично вимикає неактивні потоки від завершення роботи до завершення роботи. За замовчуванням політика підтримки активності застосовується лише тоді, коли потоків більше ніж corePoolSize. Але метод allowCoreThreadTimeOut(boolean) також можна використовувати для застосування цієї політики тайм-ауту до основних потоків, якщо значення keepAliveTime не дорівнює нулю.

Приклад -***ThreadPoolExecutor***(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue)

```
public static void testExecutor(){
   ThreadPoolExecutor executor = new ThreadPoolExecutor(
         2,  // corePoolSize
         2,  // maximumPoolSize
         10, // keepAliveTime
         TimeUnit.SECONDS,
         new ArrayBlockingQueue<>(2)  // workQueue
   );


   for (int i = 0; i < 10; i++) {
      int taskNo = i;
      executor.execute(() -> {
         System.out.println("Виконується задача: " + taskNo + " | " +
Thread.currentThread().getName());
      });
   }
   executor.shutdown();
}
```

Пул потоків: У нас є 2 основних потоки (corePoolSize = 2).
Максимальний розмір пулу: Також 2 (maximumPoolSize = 2), тобто нові потоки не створюються понад corePoolSize.
Черга задач: Використовується ArrayBlockingQueue<>(2), що означає, що 2 задачі можуть очікувати в черзі.
Що станеться при надлишку задач?
Одночасно виконується 2 задачі.
Ще 2 задачі можуть чекати в черзі.
Інші задачі блокуються, доки не звільниться місце (тобто нові execute() будуть чекати).
Очікуваний результат виконання testExecutor()
Перша пара задач (0 і 1) починає виконання в двох потоках.
Наступні 2 задачі (2 і 3) додаються в чергу.
Після завершення перших двох задач вони звільняють потоки, і задачі 2 і 3 починають виконання.
Цикл повторюється, доки всі задачі не будуть виконані.

Що буде, якщо збільшити maximumPoolSize?

ThreadPoolExecutor автоматично регулює розмір пулу (getPoolSize()) відповідно до меж, встановлених corePoolSize (див. getCorePoolSize()) і maximumPoolSize (getMaximumPoolSize()).

Коли нове завдання надсилається в метод execute(Runnable) і запущено менше потоків corePoolSize, створюється новий потік для обробки запиту, навіть якщо інші робочі потоки неактивні. Якщо запущено більше потоків ніж corePoolSize, але менше ніж maximumPoolSize, новий потік буде створено, лише якщо чергу заповнено.

Встановивши однакові corePoolSize і maximumPoolSize, ви створюєте пул потоків ***фіксованого розміру***. Встановлюючи для параметра maximumPoolSize фактично необмежене значення, наприклад Integer.MAX_VALUE, ви дозволяєте пулу розміщувати довільну кількість одночасних завдань. Зазвичай основний і максимальний розміри пулу встановлюються лише під час створення, але їх також можна змінювати динамічно за допомогою setCorePoolSize(int) і setMaximumPoolSize(int).

За замовчуванням навіть основні потоки спочатку створюються та запускаються лише тоді, коли надходять нові завдання, але це можна динамічно перевизначати за допомогою методу prestartCoreThread() або prestartAllCoreThreads().

Нові потоки створюються за допомогою ThreadFactory. Якщо не вказано інше, використовується Executors.defaultThreadFactory(), який створює потоки, щоб усі були в тій самій ThreadGroup і з однаковим пріоритетом NORM_PRIORITY і статусом не демона.

Ви можете змінити ім'я потоку, групу потоку, пріоритет, статус демона тощо.

Якщо ThreadFactory не зможе створити потік на запит, повертаючи значення null із newThread, виконавець продовжить роботу, але може не мати змоги виконати жодне завдання. Потоки повинні мати дозвіл RuntimePermission "modifyThread". Якщо робочі потоки або інші потоки, що використовують пул, не мають цього дозволу, сервіс може бути погіршений: зміни конфігурації можуть не вступити в силу своєчасно, а пул завершення роботи може залишатися в стані, у якому припинення можливе, але не завершено.

Конструктори

ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, RejectedExecutionHandler handler)

Цей конструктор схожий на попередній, проте дозволяє вказати власний RejectedExecutionHandler:

handler: Обробник, який використовується, коли виконання задачі відмовлено (наприклад, через перевантаження пулу потоків).

ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory)

Цей конструктор дозволяє вказати власний ThreadFactory - Фабрика, що використовується для створення нових потоків.

ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler)

Цей конструктор дає повний контроль над усіма параметрами:

threadFactory: Указана фабрика для створення нових потоків.

handler: Указаний обробник відмов виконання задач.

```
public static void testExecutorAbortPolicy()
{
   ThreadPoolExecutor executor = new ThreadPoolExecutor(
         2,
         2,
         10,
         TimeUnit.SECONDS,
         new ArrayBlockingQueue<>(1),

   new ThreadPoolExecutor.AbortPolicy()
   );

   try {
      for (int i = 0; i < 10; i++) {
         int taskNumber = i;
         executor.execute(() -> {
            System.out.println("Running task " + taskNumber);

            // Simulate long-running task
            try {
               Thread.sleep(1000);
            } catch (InterruptedException e) {
               Thread.currentThread().interrupt();
            }
         });
         System.out.println("Task " + taskNumber + " submitted successfully");
      }
   } catch (RejectedExecutionException e) {
      System.err.println("Task submission rejected: " + e.getMessage());
   } finally {
      executor.shutdown();
   }
}
```

Пул потоків: Все ще **2 основні потоки** (corePoolSize = 2).**Максимальний розмір пулу**: **Також 2**, тобто **нові потоки не створюються** понад corePoolSize.**Черга задач**: Тепер **тільки 1 місце в черзі** (ArrayBlockingQueue<>(1)).**Політика відмови**: AbortPolicy означає, що якщо черга і всі потоки зайняті, то **нові задачі будуть відхилені і викличуть RejectedExecutionException**.

Приклад - ***ThreadPoolExecutor***(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler)

```
public class CustomCallerRunsPolicy implements RejectedExecutionHandler {
    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        if (!executor.isShutdown()) {
            System.out.println("Executing task in caller thread: " + Thread.currentThread().getName());
            r.run();  // Виконуємо задачу в потоці, що викликав `execute()`
        }
    }
}

    ThreadPoolExecutor executor = new ThreadPoolExecutor(
            2,
            4,
            5,
            TimeUnit.SECONDS,
            new ArrayBlockingQueue<>(2),
            threadFactory,
            new CustomCallerRunsPolicy()
    );

    for (int i = 0; i < 15; i++) {
        int taskNo = i;
        executor.execute(() -> {
            System.out.println("Виконується задача: " + taskNo + " | " +
Thread.currentThread().getName());
        });
    }
    executor.shutdown();
}
```

ThreadPoolExecutor параметри:
corePoolSize = 2: спочатку створюється **2 потоки**.
maximumPoolSize = 4: якщо черга заповнена, додаються **ще до 2 потоків**.
keepAliveTime = 5 секунд: додаткові потоки завершуються після 5 секунд без роботи.
workQueue = ArrayBlockingQueue<>(2): черга з 2 задачами.
Кастомний ThreadFactory:
Створює потоки з іменем CustomThread-ID.
CallerRunsPolicy як RejectedExecutionHandler:
Якщо **черга переповнена** і немає вільного потоку, **задача виконується у потоці виклику**

Приклад - ***ThreadPoolExecutor***(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler)

```
public class CustomDiscardPolicy implements
RejectedExecutionHandler {
    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor
executor) {
        System.out.println("Task discarded: " + r.toString());
        // Завдання просто не додається, нічого не робимо
    }
}
```

```
    ThreadPoolExecutor executor = new ThreadPoolExecutor(
            2,
            4,
            5,
            TimeUnit.SECONDS,
            new ArrayBlockingQueue<>(2),
            threadFactory,
            new CustomDiscardPolicy ()
    );

    for (int i = 0; i < 15; i++) {
        int taskNo = i;
        executor.execute(() -> {
            System.out.println("Виконується задача: " + taskNo + " | " +
Thread.currentThread().getName());
        });
    }
    executor.shutdown();
}
```

ThreadPoolExecutor параметри:
corePoolSize = 2: спочатку створюється **2 потоки**.
maximumPoolSize = 4: якщо черга заповнена, додаються **ще до 2 потоків**.
keepAliveTime = 5 секунд: додаткові потоки завершуються після 5 секунд без роботи.
workQueue = ArrayBlockingQueue<>(2): черга з 2 задачами.
Кастомний ThreadFactory:
Коли пул потоків і черга заповнені, **нова задача просто ігнорується**.**Немає помилок або винятків**, але є **втрата задач**.Можна додати логування або спеціальний обробник.

Приклад - ***ThreadPoolExecutor***(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler)

```
public static class CustomDiscardOldestPolicy implements RejectedExecutionHandler {
    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        if (!executor.isShutdown()) {
            Runnable oldestTask = executor.getQueue().poll(); // Видаляємо найстарішу задачу
            System.out.println("Discarding oldest task: " + (oldestTask != null ? oldestTask.toString() :
"None"));
            executor.execute(r); // Пробуємо ще раз додати нову задачу
        }
    }
}

ThreadPoolExecutor executor = new ThreadPoolExecutor(
            2,
            4,
            5,
            TimeUnit.SECONDS,
            new ArrayBlockingQueue<>(2),
            threadFactory,
            new CustomDiscardOldestPolicy ()
    );

    for (int i = 0; i < 15; i++) {
        int taskNo = i;
        executor.execute(() -> {
            System.out.println("Виконується задача: " + taskNo + " | " +
Thread.currentThread().getName());
        });
    }
    executor.shutdown();
}
```

ThreadPoolExecutor параметри:
corePoolSize = 2: спочатку створюється **2 потоки**.
maximumPoolSize = 4: якщо черга заповнена, додаються **ще до 2 потоків**.
keepAliveTime = 5 секунд: додаткові потоки завершуються після 5 секунд без роботи.
workQueue = ArrayBlockingQueue<>(2): черга з 2 задачами.
Кастомний ThreadFactory:
Якщо черга повна, беремо **найстарішу задачу (poll()) і видаляємо її**.Потім **ще раз пробуємо додати нову задачу** (executor.execute(r);).Якщо executor закритий – просто нічого не робимо.

Постановка в чергу

Будь-яку чергу BlockingQueue можна використовувати для передачі та утримання надісланих завдань.

Існує три загальні стратегії черги:

Прямі передачі (*Direct handoffs)*. Хорошим вибором за замовчуванням для робочої черги є ***SynchronousQueue*** - спроба поставити завдання в чергу буде невдалою, якщо жоден потік не буде негайно доступний для його виконання, тому буде створено новий потік. Ця політика дозволяє уникнути блокувань під час обробки наборів запитів, які можуть мати внутрішні залежності. Прямі передачі зазвичай вимагають необмежених максимальних розмірів пулу, щоб уникнути відхилення нових поданих завдань. Це, у свою чергу, допускає можливість необмеженого зростання потоку, коли команди продовжують надходити в середньому швидше, ніж вони можуть бути оброблені.

Необмежені черги (*Unbounded queues)*. Використання необмеженої черги (наприклад, ***LinkedBlockingQueue*** без попередньо визначеної ємності) призведе до очікування нових завдань у черзі, коли всі потоки corePoolSize зайняті. Таким чином, ніколи не буде створено більше потоків corePoolSize. (Тому значення maximumPoolSize не має жодного ефекту.)

Необмежені черги можуть бути доцільними, коли кожне завдання є повністю незалежним від інших, тому завдання не можуть впливати на виконання одне одного; наприклад, на сервері веб-сторінок. Хоча цей стиль черги може бути корисним для згладжування тимчасових спалахів запитів, він допускає можливість необмеженого зростання робочої черги, коли команди продовжують надходити в середньому швидше, ніж вони можуть бути оброблені.

Приклад - Прямі передачі (*Direct handoffs)*

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(
        2, // corePoolSize
        4, // maximumPoolSize
        60L, // keepAliveTime
        TimeUnit.SECONDS, // unit of keepAliveTime
        new SynchronousQueue<>(), // no capacity
        new ThreadPoolExecutor.AbortPolicy() // Rejection policy that aborts
    );

    // Introduce tasks
    try {
        for (int i = 0; i < 8; i++) {
            final int taskNumber = i;
            System.out.println("Submitting task " + taskNumber);
            executor.execute(() -> {
                try {
                    System.out.println("Running task " + taskNumber);
                    Thread.sleep(3000); // Make the thread busy
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }
    } catch (RejectedExecutionException e) {
        System.err.println("Task rejected: " + e.getMessage());
    }

    executor.shutdown();
}
```

Direct Handoffs (Прямий передавальний механізм)
Завдання **не ставляться в чергу**. Якщо є вільний потік, **задача передається безпосередньо** йому. Якщо всі потоки зайняті – **задача відхиляється** (RejectedExecutionException).

Обмежені черги *Bounded queues*. Обмежена черга (наприклад, ArrayBlockingQueue) допомагає запобігти виснаженню ресурсів при використанні з обмеженими максимальними розмірами пулу, але її може бути складніше налаштувати та контролювати. Розміри черги та максимальні розміри пулу можуть бути замінені один на одного: використання великих черг і малих пулів мінімізує використання ЦП, ресурсів ОС і витрати на перемикання контексту, але може призвести до штучно низької пропускної здатності. Якщо завдання часто блокуються (наприклад, якщо вони прив’язані до вводу-виводу), система може мати змогу запланувати час для більшої кількості потоків, ніж ви дозволяєте. Використання малих черг, як правило, потребує більших розмірів пулу, що сприяє завантаженню ЦП, але може призвести до неприйнятних накладних витрат на планування, що також зменшує пропускну здатність.

Відхилені завдання (**Rejected tasks)**

Нові завдання, надіслані в методі execute(Runnable), будуть відхилені, коли Executor буде вимкнено, а також коли Executor використовує обмежені межі як для максимальних потоків, так і для ємності робочої черги, і він насичений. У будь-якому випадку метод execute викликає метод RejectedExecutionHandler.rejectedExecution(Runnable, ThreadPoolExecutor) свого RejectedExecutionHandler.

Надаються чотири попередньо визначені політики обробки:

ThreadPoolExecutor.AbortPolicy за замовчуванням обробник кидає RejectedExecutionException часу виконання після відхилення. ThreadPoolExecutor.CallerRunsPolicy потік, який викликає виконання, сам запускає завдання. Це забезпечує простий механізм керування зворотним зв’язком, який уповільнює швидкість надсилання нових завдань.

ThreadPoolExecutor.DiscardPolicy завдання, яке неможливо виконати, просто відкидається.

ThreadPoolExecutor.DiscardOldestPolicy, якщо виконавець не вимикається, завдання на початку робочої черги відкидається, а потім виконується повторна спроба (яка знову може завершитися помилкою, спричиняючи повторення). Можна визначати та використовувати інші типи класів RejectedExecutionHandler. Це вимагає певної обережності, особливо якщо політики розроблено для роботи лише за певної потужності або політики черги.

```
import java.util.concurrent.*;

class CustomRejectedHandler implements RejectedExecutionHandler {
   @Override
   public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
      System.out.println("Task rejected: " + r.toString());

   }
}

public class CustomRejectedTaskExecutor {
   public static void main(String[] args) {
      ThreadPoolExecutor executor = new ThreadPoolExecutor(
         2, 2, 10L, TimeUnit.SECONDS,
         new ArrayBlockingQueue<>(2),
         new CustomRejectedHandler()
      );

      for (int i = 0; i < 6; i++) {
         int taskNo = i;
         executor.execute(() -> {
            System.out.println("Executing task " + taskNo);
            try { Thread.sleep(2000); } catch (InterruptedException e) { }
         });
      }

      executor.shutdown();
   }
}
```

3. Rejected Tasks (Обробка відхилених задач)
Використовується кастомний RejectedExecutionHandler.
Якщо задача відхиляється, її можна:
Виконати в основному потоці (CallerRunsPolicy).
Логувати / повторити пізніше.

Очікувана поведінка:
2 задачі виконуються одразу.
2 задачі чекають у черзі.
Наступні 2 задачі будуть відхилені та залоговані кастомним RejectedExecutionHandler.

public interface Executor

Executor — це інтерфейс у пакеті java.util.concurrent, який визначає єдиний метод execute(), призначений для асинхронного виконання задач. Він є основою для багатьох механізмів керування потоками у Java.
Об’єкт, який виконує надіслані завдання Runnable. Цей інтерфейс забезпечує спосіб відокремлення подання завдання від механізму виконання кожного завдання, включаючи деталі використання потоку, планування тощо. Зазвичай замість явного створення потоків використовується Виконавець. Наприклад, замість того, щоб викликати new Thread(new(RunnableTask())).start() для кожного з набору завдань, ви можете використовувати: Виконавець виконавець = anExecutor; executor.execute(нове RunnableTask1()); executor.execute(нове RunnableTask2());

Основна ідея Executor

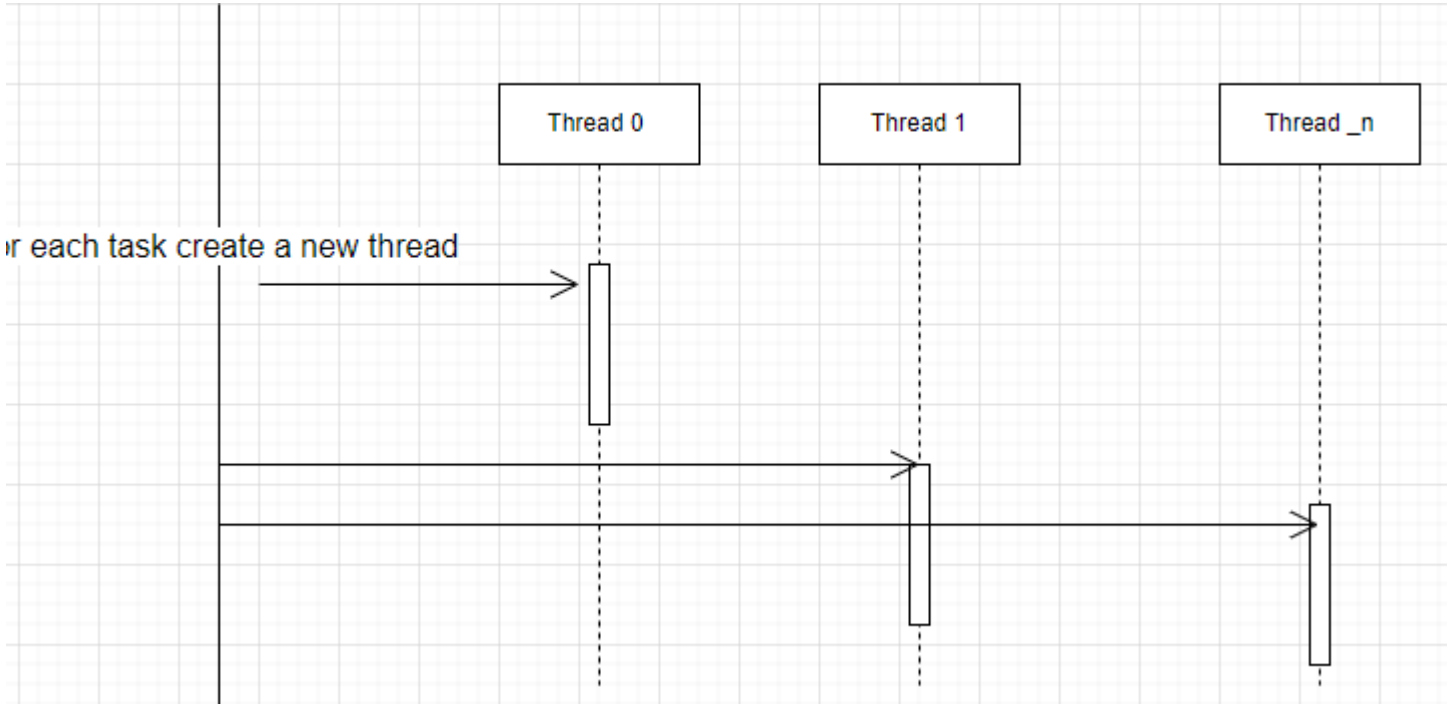
Замість того щоб напряму створювати і керувати потоками (Thread), інтерфейс Executor дозволяє передавати завдання (Runnable) для виконання спеціалізованим механізмам, які самостійно керують потоками.

Пул потоків — це режим використання об'єднаного потоку. Створивши певну кількість потоків, можна підвищити швидкість відповіді системи, утримуючи ці потоки в стані готовності. Після використання потоків вони повертаються до пулу потоків. Таким чином, досягається повторне використання потоку, що зменшує споживання системних ресурсів.

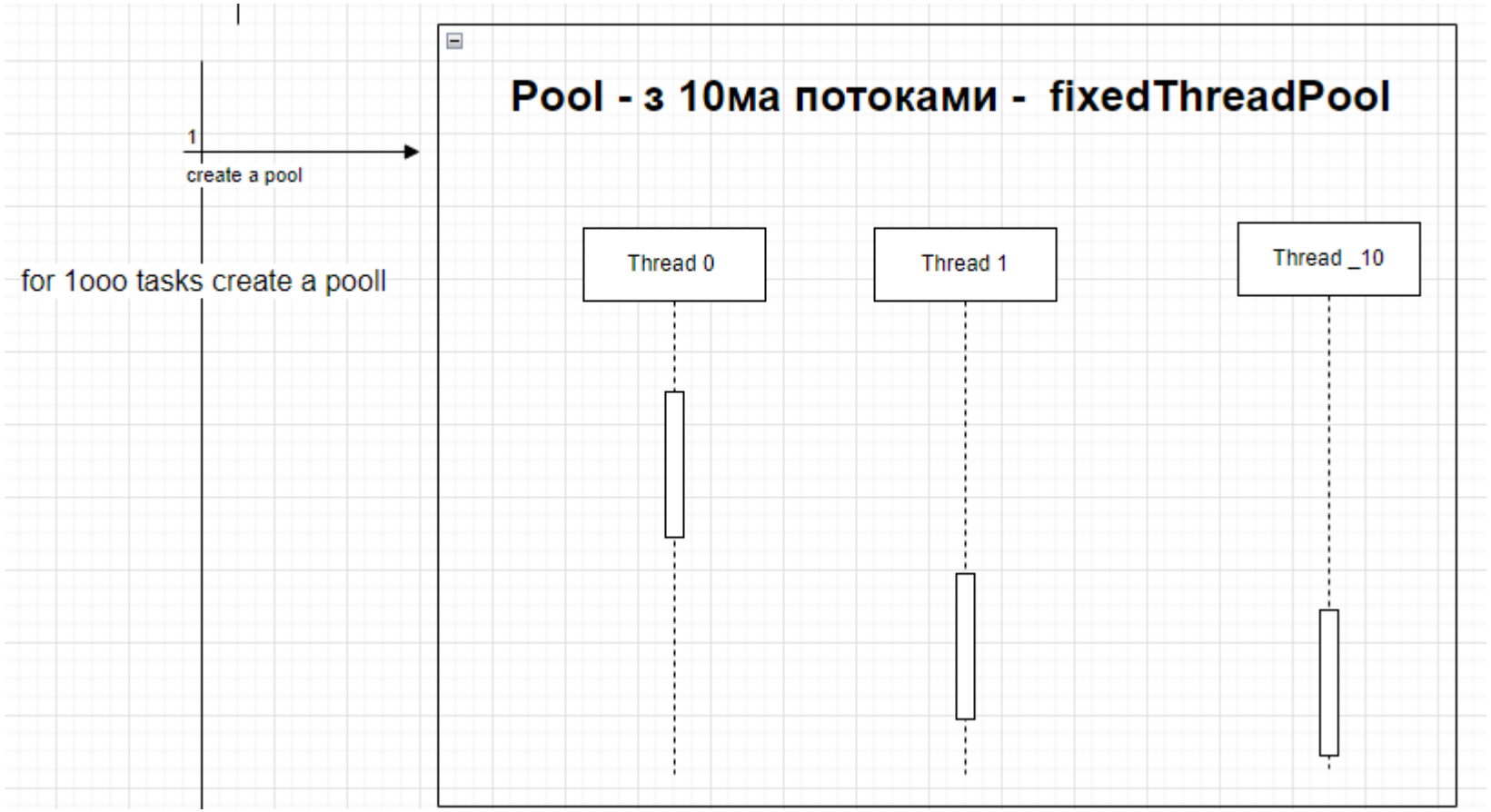
Приклад створення ***10*** потоків.

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 10; i++) {
            Thread mythread = new Thread(new SimpleTask());
            mythread.start();
        }

public class SimpleTask implements Runnable {
    @Override
    public void run(){
        System.out.println(Thread.currentThread().getName());
    }
}
```

Java для кожної задачі створює потік і після виконання його закриває (shutdown).
Проблема - багато машинних ресурсів



Рішення проблеми – створити pool, який включатиме певну кількістю потоків. Наприклад для 1000 задач – Маємо пул з 10 потоків

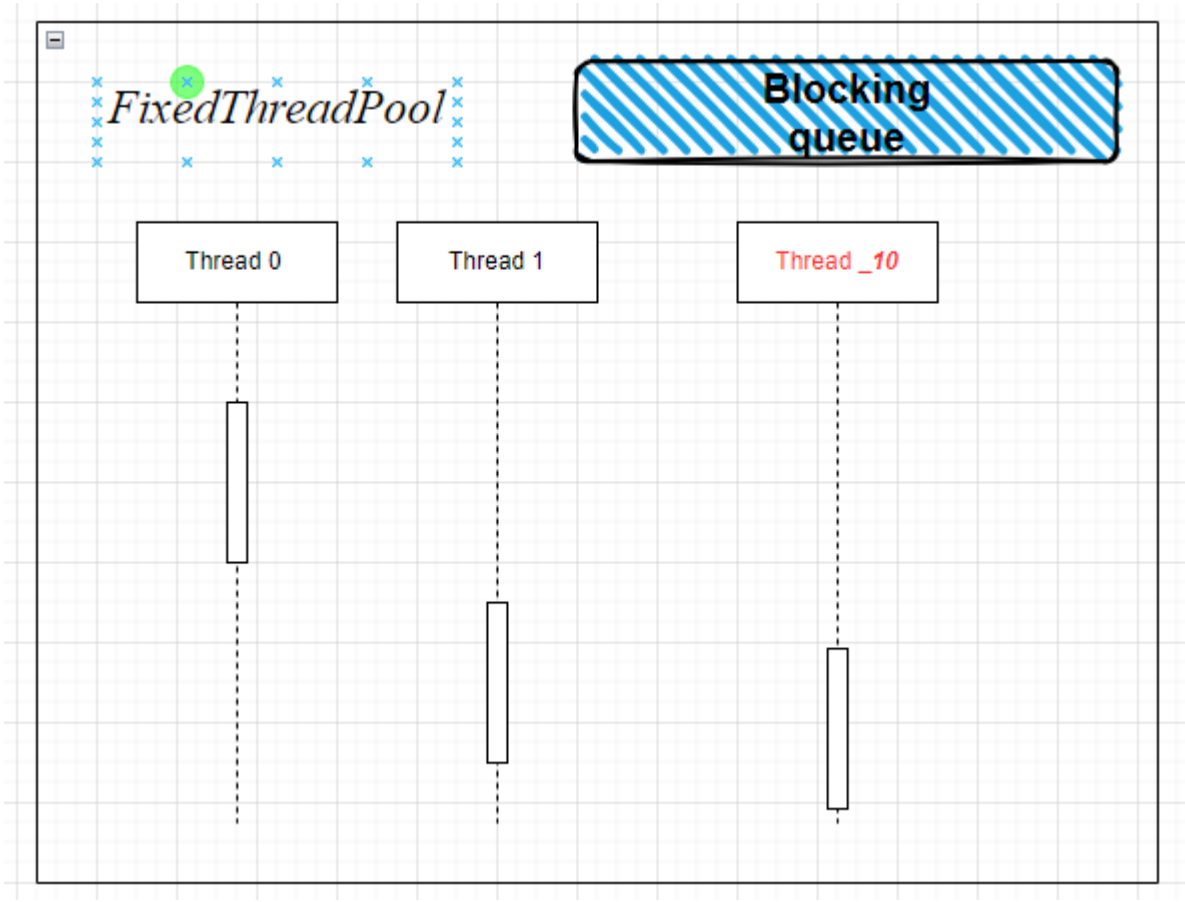


Приклад - *Executors.newFixedThreadPool(10). Замість 1000 потоків створюються 10. Для виконання задачі пул чекає поки якийсь з 10ти потоків стане вільний і передає йому задачу на виконання. Виклик* **service.shutdown() – ініціює закриття потоків.**

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService service = Executors.newFixedThreadPool(6);
        for (int i = 0; i < 1000; i++) {
            service.execute(new SimpleTask());
        }
       service.shutdown();
}
public class SimpleTask implements Runnable {
    @Override
    public void run(){
        System.out.println(Thread.currentThread().getName());
    }
}
```

Перший тип пулів - FixedThreadPool

FixedThreadPool – розміщує задачі для виконання в «blocking queue», і кожен потік виконує 2 операції: отримати задачу з черги; виконати задачу. Якщо всі потока зайняті - пул чекає на вільний потік.



Переваги

1) Зменшення споживання ресурсів: пули потоків можуть зменшити споживання, викликане створенням і знищенням потоку, шляхом повторного використання створених потоків.

2) Покращення швидкості виконання завдань потоку.

3) Покращення керованості потоками: потоки є дефіцитними ресурсами, а необмежене створення потоків споживає системні ресурси та знижує стабільність системи. Пули потоків можна використовувати для уніфікованого розподілу, налаштування та моніторингу.

Недоліки

Як визначити необхідну кількість потоків?

Один із методів – це визначати кількість потоків залежно від «обчислювальних» ядер в центральному процесорі.

Приклад - Runtime.getRuntime().availableProcessors();

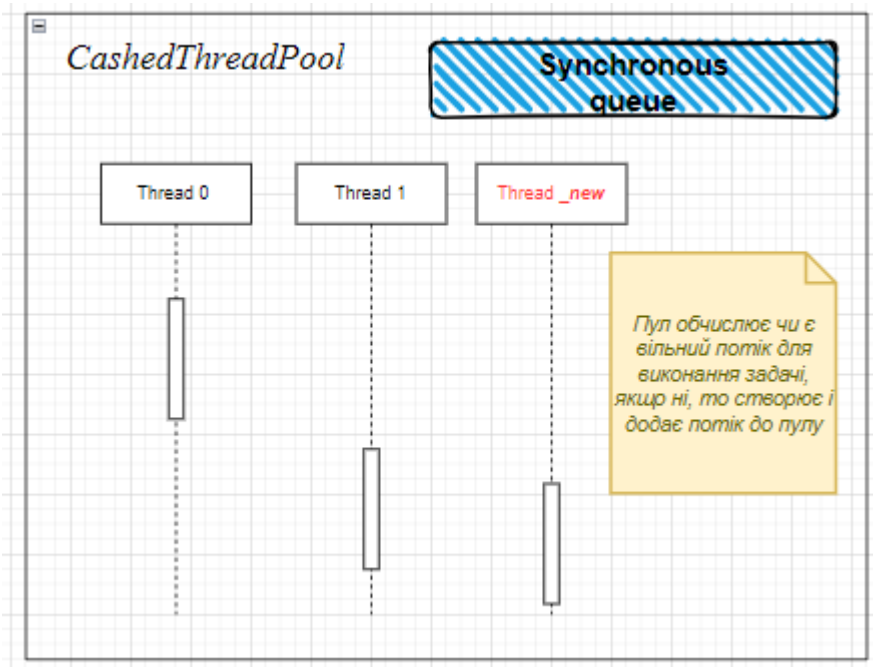
```
public class Main {
    public static void main(String[] args) throws InterruptedException {
int numcore = Runtime.getRuntime().availableProcessors();
ExecutorService service = Executors.newFixedThreadPool(numcore);

        for (int i = 0; i < 1000; i++) {
            service.execute(new SimpleTask());
        }
}
public class SimpleTask implements Runnable {
    @Override
    public void run(){
        System.out.println(Thread.currentThread().getName());
    }
}
```

Такий підхід не працює, коли багато задач типу - Input/Output.

Другий тип пулів - CashedThreadPool

Рішення проблеми – створити pool без заздалегідь визначеної кількості потоків – CashedThreadPool.

Такий пул має Synchronous queue, яка розрахована на 1 задачу тільки. Кожен раз коли пул отримує задачу – Пул шукає на «вільний» потік. Якщо всі потоки «зайняті», тоді пул створює новий потік і додає його до пулу. Якщо в пулі є потік, який не виконує задачі протягом 60sec –такий потік видаляється з пулу.



Приклад – кількість потоків для 1000 задач буде визначатись пулом

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
ExecutorService service = Executors.newCachedThreadPool();

        for (int i = 0; i < 1000; i++) {
            service.execute(new SimpleTask());
        }
}
public class SimpleTask implements Runnable {
    @Override
    public void run(){
        System.out.println(Thread.currentThread().getName());
    }
}
```

Третій тип пулів - ScheduledThreadPool

ScheduledThreadPool – дозволяє визначити інтервали часу з яким задачі будуть виконуватись пулом, за допомогою методів

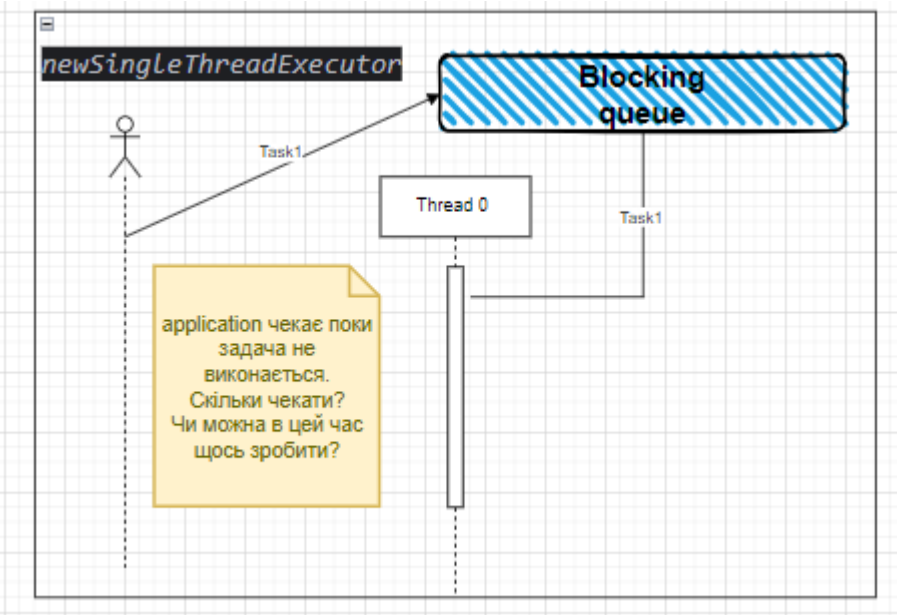
Service.schedule – визначає проміжок часу, через який задача буде виконана після закінчення попередньої.

Service.scheduleAtFixedRate – дозволяє задати час очікування перед запуском першої задачі, і потім проміжок часу через який буде взято наступну задачу, незалежно від того завершилось виконання попередньої задачі чи ні

Service.scheduleWithFixedDelay - дозволяє задати час очікування перед запуском першої задачі, і потім проміжок часу через який буде взято наступну задачу, після завершення попередньої задачі.

```
private static void schedulepoolfixedrate(){
    ScheduledExecutorService service = Executors.newScheduledThreadPool(2);

    Runnable taskAtFixedRate = () -> {
        String execTime = new SimpleDateFormat("mm:ss.SSS").format(new Date());
        System.out.println("Task with scheduleAtFixedRate executed at: " + execTime);

    };
    service.scheduleAtFixedRate(taskAtFixedRate, 0, 1, TimeUnit.SECONDS); // Execute every 1 second

    Runnable taskWithFixedDelay = () -> {
        String execTime = new SimpleDateFormat("mm:ss.SSS").format(new Date());
        System.out.println("Task with scheduleWithFixedDelay executed at: " + execTime);
        try {
            Thread.sleep(3000); // Simulating task execution time
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    };
    service.scheduleWithFixedDelay(taskWithFixedDelay, 0, 2, TimeUnit.SECONDS);

    service.schedule(() -> {
        service.shutdown();
    }, 10, TimeUnit.SECONDS);
}
```

***Четвертий тип пулів* – SingleThreaded (схожий на *FixedThreadPool*) при цьому пул включає тільки 1 потік. Задачі такого пулу виконуються завжди послідовно.**



Таким чином програма, щоб не простоювати поки пул виконує задачі, має якимось чином вміти визначити: виконана задача чи ні?

Interface Runnable – описує інтерфейс, який не повертає результати
Interface Callable - – описує інтерфейс, який повертає результати. Єдиний метод call() – не має аргументів. Виняткова ситуація можлива, якщо метод call() не може обчислити результати.
Залежно від того, яким типом буде визначено genric V – такого типу результат буде повернено.

```
@FunctionalInterface
public interface Runnable {
   /**
    * Runs this operation.
    */
   void run();
}
```

```
@FunctionalInterface
public interface Callable<V> {
   /**
    * Computes a result, or throws an exception if unable to do so.
    *
    * @return computed result
    * @throws Exception if unable to compute a result
    */
   V call() throws Exception;
}
```

```
public class MyCallable implements Callable<String>{
   String runName;
   public MyCallable(String runName){
      this.runName=runName;
   }
   public String call() throws Exception{

         try {
            Thread.sleep(100);
         } catch (InterruptedException e) {
            throw new RuntimeException(e);
         }
      return "Thread: "+Thread.currentThread().getName() +",
completed: "+runName;
   }
}
```

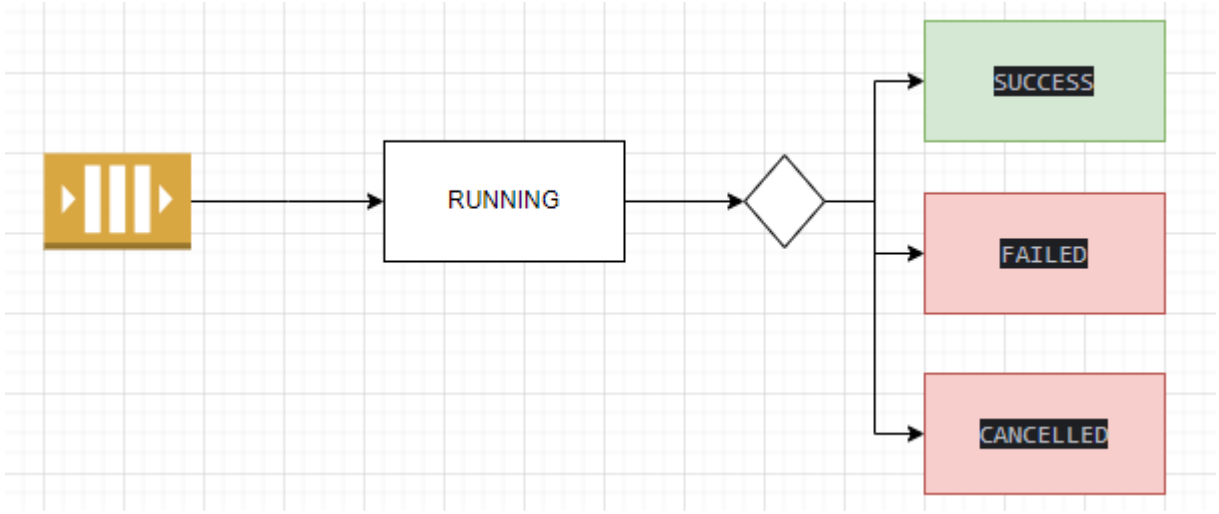
```
public class MyCallable implements Callable<String>{
   String runName;
   public MyCallable(String runName){
      this.runName=runName;
   }
   public String call() throws Exception{

         try {
            Thread.sleep(100);
         } catch (InterruptedException e) {
            throw new RuntimeException(e);
         }
      return "Thread:
"+Thread.currentThread().getName() +", completed:
"+runName;
   }
}
```

```
ExecutorService service =
Executors.newSingleThreadExecutor();
Future <String> poolFuture = service.submit(new
MyCallable("mytask"));

try {
   String result = poolFuture.get(500,
TimeUnit.MICROSECONDS);
   System.out.println("results "+result);

} catch (InterruptedException e) {
   throw new RuntimeException(e);
} catch (ExecutionException e) {
   throw new RuntimeException(e);
}catch (TimeoutException e) {
   System.out.println("Could not complete the task
before time out");
}

service.shutdown();
```

Interface Future –включає методи для опису поточного стану задачі, яка була надіслана на виконання

Життєвий цикл задачі

- ✓ ***Задача в черзі;***
- ✓ ***Задача на виконанні - RUNNING;***
- ✓ ***Задача виконана (SUCCESS);***
- ✓ ***Задача не виконана через неможливість обчислити (FAILED)***
- ✓ ***Задача не виконана через виклик «cancelled» (причина – закриття потоку)***



Методи інтерфейсу Future для визначення кожного відповідного стану задачі

- ✓ ***isDone()** – повертає True, якщо задачу завершено без виняткової ситуації.*
- ✓ ***isCancelled()** - повертає True, якщо задачу завершено без виняткової ситуації або з винятковою ситуацією*
- ✓ ***Get(long timeout, TimeUnit unit)** – чекає протягом заданого проміжку часу, потім*

Якщо задачу виконано – completed, якщо не виконано, тоді:

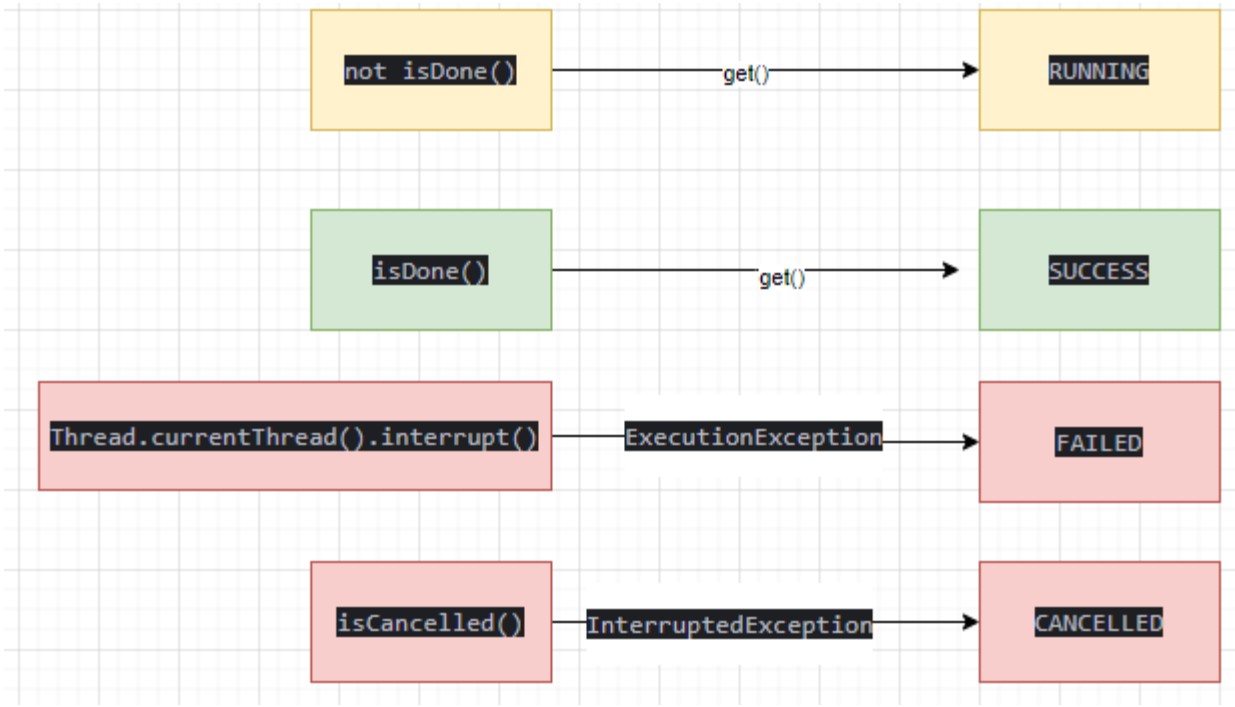
***CancellationException** – задачу скасовано.*

***ExecutionException** – помилка при обчисленні*

***InterruptedException** –*

***TimeoutException** – задача не обчислена протягом заданого часу*

- ✓ **resultNow()** - *повертає поточний стан задачі, на поточний момент, не чекає.*



Приклад - до тих пір поки, задача в стані RUNNING – чекаємо. Поточний стан визначаємо методом get()

```
private static void singlethreadsingletask() {

    ExecutorService service = Executors.newSingleThreadExecutor();
    Future <String> poolFuture = service.submit(new MyCallable("mytask"));

    while (!poolFuture.isDone()){

        System.out.println("waiting for future");
    }
    try {
        System.out.println(poolFuture.get());
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    } catch (ExecutionException e) {
        throw new RuntimeException(e);
    }
    service.shutdown();

}
```

Приклад – В стані RUNNING 10 задач. Відслідковуємо Поточний стан визначаємо методом get() для кожної.

```
private static void singlethreadmultitask() {

    ExecutorService service = Executors.newSingleThreadExecutor();
    List<Future> allFutures = new ArrayList<>();
    for (int i =0; i<10; i++) {
        Future<String> poolfuture = service.submit(new MyCallable("mytask"));
        allFutures.add(poolfuture);
    }
    for (int i =0; i<10; i++) {
        Future future = allFutures.get(i);

        try {
            System.out.println(future.get());
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        } catch (ExecutionException e) {
            throw new RuntimeException(e);
        }
    }
    service.shutdown();
}
```

Приклад – В стані RUNNING 10 задач. Відслідковуємо Поточний стан визначаємо методом get(long timeout, TimeUnit unit) для кожної.

```
private static void singlethreadsingletasktimeout() {

   ExecutorService service = Executors.newSingleThreadExecutor();
   Future <String> poolFuture = service.submit(new MyCallable("mytask"));

   try {
      String result = poolFuture.get(500, TimeUnit.MICROSECONDS);
      System.out.println("results "+result);

   } catch (InterruptedException e) {
      throw new RuntimeException(e);
   } catch (ExecutionException e) {
      throw new RuntimeException(e);
   }catch (TimeoutException e) {
      System.out.println("Could not complete the task before time out");
   }

   service.shutdown();

}
```

При створенні екземпляру пулу, викликається конструктор, який має набір параметрів для всі 4х типів пулі.

Executors.*newFixedThreadPool*(2)

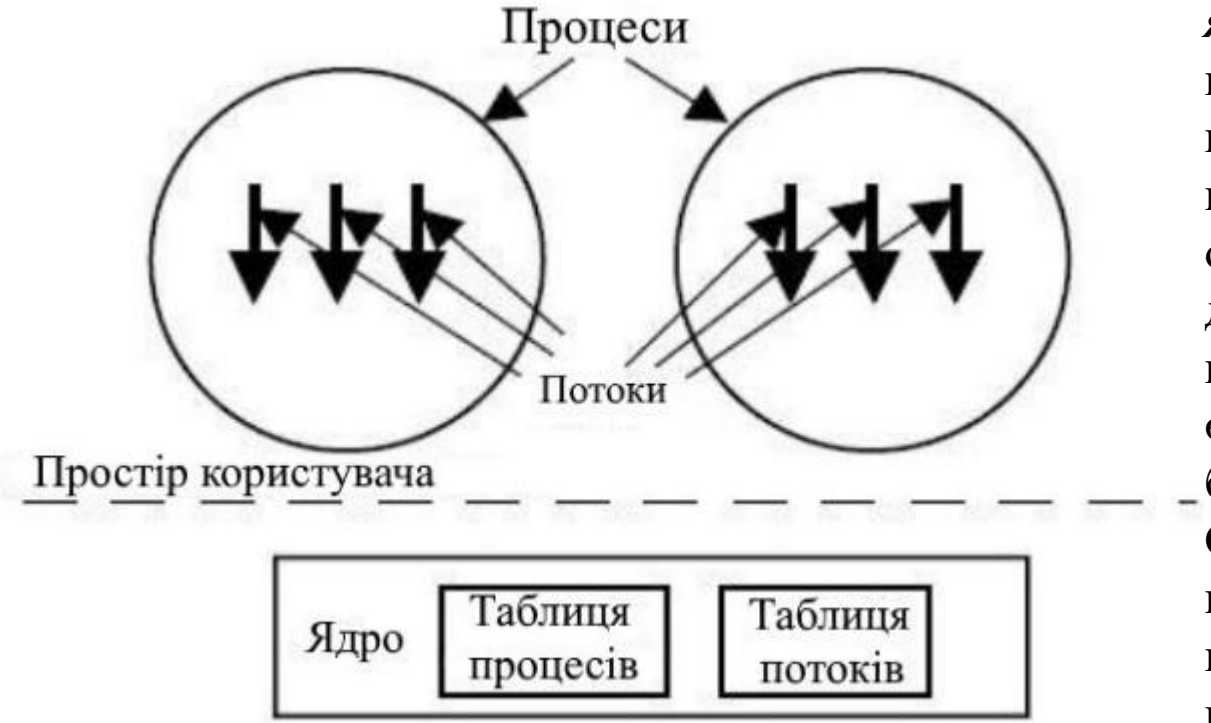

```
public ThreadPoolExecutor(int
corePoolSize,
int maximumPoolSize,
long keepAliveTime,
TimeUnit timeUnit,
BlockingQueue<Runnable> workQueue,
ThreadFactory threadFactory,
RejectedExecutionHandler handler)
```

- ✓ corePoolSize – кількість «kernel» потоків в полі, які є ***активними за замовченням***, ніколи не видаляються з пулу. ***Що таке «kernel» потік?***
- ✓ ***Як визачається розмір пулу?***
- ✓ corePoolSize+Current Pool size (додані/видалені потоки).
- ✓ maximumPoolSize – максимальна кількість потоків, яка може бути включена в пул.
- ✓ keepAliveTime – час протягом, якого пул може не мати задачі (тобто бути idle) до момент коли його бути «терміновано». Якщо потік без задачі довше за визначений час і цей потік не є «kernel» тоді його буде перевикористано або закрито. «kernel» потік - ніколи не «закривається». Якщо параметр allowCoreThreadTimeout визначено «true» тоді «kernel» потік також можна перевикористати.

Порівняльна таблиця

Параметр	FixedThreadPool	CachedThreadPool	ScheduledThreadPool	SingleThreaded
corePoolSize	Передається, як параметр	0	Передається, як параметр	1
maximumPoolSize	Дорівнює corePoolSize	Integer.MAX_VALUE	Integer.MAX_VALUE	1
keepAliveTime	0	60	60	0

Виділяють дві загальні категорії потоків: ***потоки на рівні користувача (UserLevel Threads – ULT) і потоки на рівні ядра (Kernel-Level Threads – KLT)***. Потоки другого типу називаються потоками, підтримуваними ядром або полегшеними (легковагими) процесами

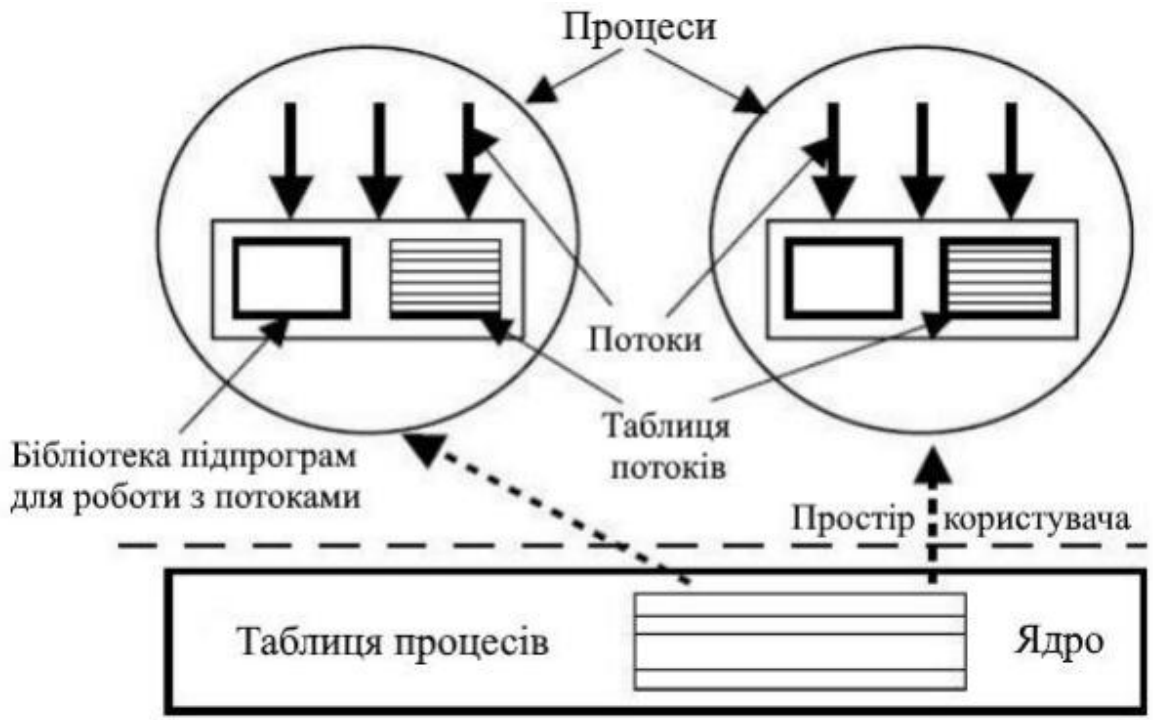


Потоки - в адресному просторі ядра

стратегія використання потоків на рівні ядра. ядро може одночасно здійснювати планування роботи декількох потоків одного і того ж процесу на декількох процесорах. По-друге, при блокуванні одного із потоків процесу ядро може вибрати для виконання інший потік цього ж процесу. Ще однією перевагою такого підходу є те, що самі процедури ядра можуть бути багатопоточними.

Основним недоліком підходу з використанням потоків на рівні ядра, в порівнянні з використанням потоків на рівні користувача, є те, що для передачі управління від одного потоку до іншого в рамках одного і того ж процесу доводитися перемикатися в режим ядра

Виділяють дві загальні категорії потоків: ***потоки на рівні користувача (UserLevel Threads – ULT) і потоки на рівні ядра (Kernel-Level Threads – KLT)***. Потоки другого типу називаються потоками, підтримуваними ядром або полегшеними (легковагими) процесами



стратегія використання потоків на рівні користувача

Для перемикання потоків не потрібно переходити в режим ядра, основної структури дані по управлінню потоками знаходяться в адресному просторі одного і того ж процес.

Потоки - в адресному просторі одного і того ж процесу.

- ✓ TimeUnit – визначає одиниці виміру для часу, коли потік може бути без задачі TimeUnit.MILLISECONDS, TimeUnit.SECONDS, and TimeUnit.MINUTES.

- ✓ BlockingQueue – черга задач, визначається типами
  - ➢ ArrayBlockingQueue – обмежена черга на основі масиву, яка надає доступ до задач за принципом FIFO.
  - ➢ LinkedBlockingQueue – умовно обмежена черга на основі списку, яка надає доступ до задач за принципом FIFO. Якщо розмір не визначено, тоді максимальний розмір Integer.MAX_VALUE
  - ➢ PriorityBlockingQueue – не обмежена черга, в які задачі відсортовані.
  - ➢ DelayQueue – не обмежена черга, яка надає доступ до задач, за умови, що час початку задачі почався з урахування задано delay time.
  - ➢ SynchronousQueue - Черга блокування, яка не зберігає елементи. У SynchronousQueue кожна операція вставки повинна чекати відповідної операції видалення іншим потоком і навпаки.
  - ➢ LinkedTransferQueue: необмежена черга на основі списків. У порівнянні з LinkedBlockingQueue, має методи transfer() і tryTranfer(), які негайно передають задачу потоку, який очікує на його отримання.
  - ➢ LinkedBlockingDeque: необмежена черга на основі deque. Це дозволяє додавати та видаляти задачі як з голови, так і з хвоста черги.

Порівняльна таблиця

Тип пулу	Тип черги	Опис
FixedThreadPool	LinkedBlockingQueue	Черга, включає необмежену кількість задач
SingleThreadExecutor	LinkedBlockingQueue	
CachedThreadPool	SynchronousQueue	
ScheduledThreadPool	DelayWorkQueue	
Custom pool (пул, який створюється викликом конструктора ThreadPoolExecutor)	ArrayBlockingQueue	

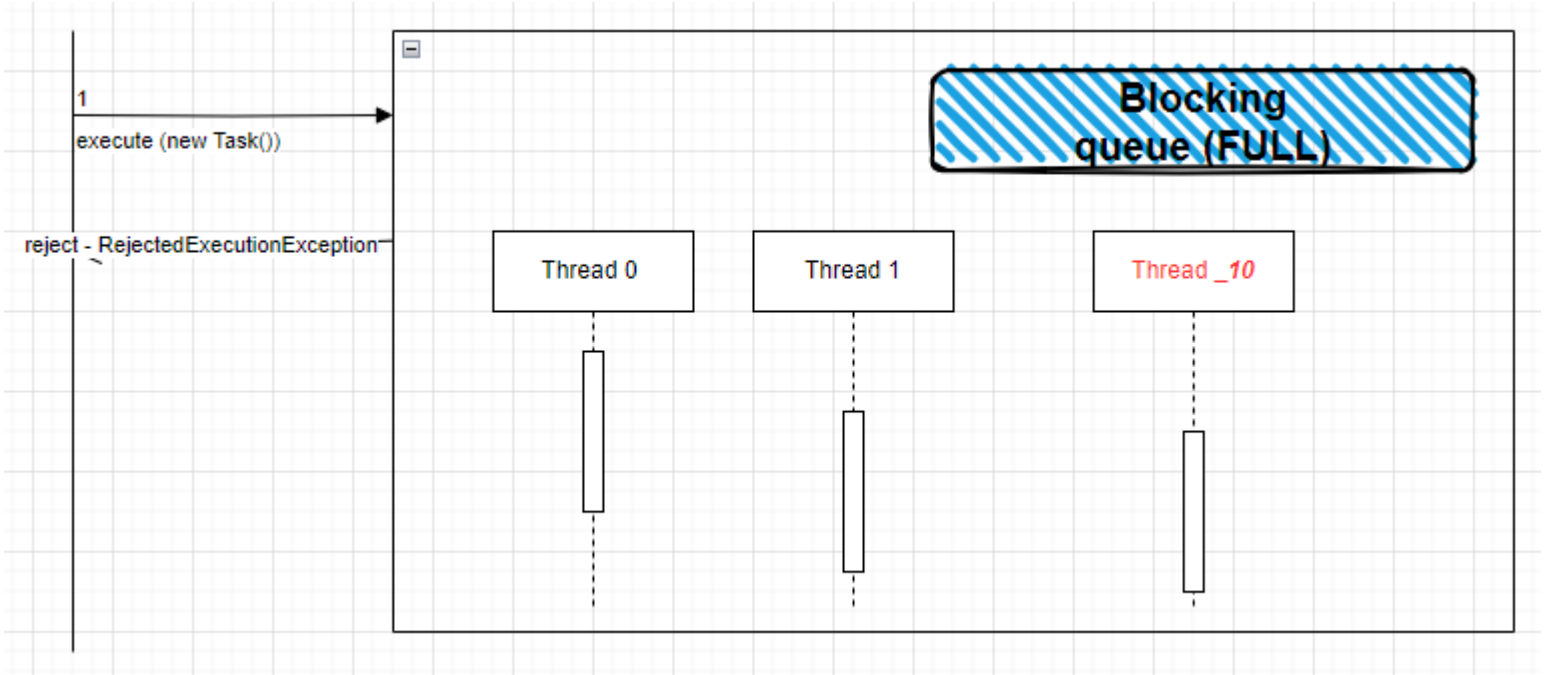
- ✓ ThreadFactory - використовується для створення нових потоків на вимогу.
- ✓ RejectedExecutionHandler- реалізує механізм відхилення, коли черга заповнена, або досягнена максимальну кількість потоків.

ThreadPoolExecutor.AbortPolicy: якщо **черга задач заповнена та не має вільного потоку** – надсилає rejected tasks, throws a RejectedExecutionException

· **ThreadPoolExecutor.DiscardPolicy:** якщо **черга задач заповнена та не має вільного потоку – ігнорує задачу.**

· **ThreadPoolExecutor.CallerRunsPolicy:** якщо **черга задач заповнена та не має вільного потоку – виконує задачу в потоці який задачу надсилає.**

· **ThreadPoolExecutor.DiscardOldestPolicy:** якщо **черга задач заповнена та не має вільного потоку – викидає з пулу «стару задачу», додає нову задачу.**



Приклад - **ThreadPoolExecutor.AbortPolicy** – визначено за замовченням.
створюємо Custom пул: один потік, черга з 1ю задачею

```
ExecutorService service = new ThreadPoolExecutor(1, 1, 1, TimeUnit.SECONDS, new
ArrayBlockingQueue<>(1),new ThreadPoolExecutor.AbortPolicy());

try {
   for (int i = 0; i < 10; i++) {
      final int taskNumber = i;
      service.execute(new SimpleTask());
      System.out.println("Task " + taskNumber + " completed");
   }

} catch (RejectedExecutionException e) {
   System.err.println("task rejected "+ e.getMessage());

}

service.shutdown();
```

Що очікуєте побачити в консолі?

Приклад - ThreadPoolExecutor.DiscardPolicy –видаляє задачу з черги. Future.state задачі - Cancelled

```
private static void rejectpolicy2() {
   ThreadPoolExecutor executor = new ThreadPoolExecutor(1,1,1,
         TimeUnit.MILLISECONDS, // time unit for keepAliveTime
         new ArrayBlockingQueue<>(1), // bounded queue with size 1
         new ThreadPoolExecutor.DiscardPolicy()); // DiscardPolicy
   List<Future> allFutures = new ArrayList<>();
   // Submit tasks to the executor
   for (int i = 1; i <= 5; i++) {
      final int taskNumber = i;
      executor.submit(() -> {
         System.out.println("Task " + taskNumber + " is running");
         try {
            Thread.sleep(1000); // Simulate task execution
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
         System.out.println("Task " + taskNumber + " completed");
      });
   }

   // Shutdown the executor
   executor.shutdown();
}
```

Що очікуєте побачити в консолі?

Приклад - **ThreadPoolExecutor. DiscardOldestPolicy()**

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(1,1,0L,
TimeUnit.MILLISECONDS, new ArrayBlockingQueue<>(1), new
ThreadPoolExecutor.DiscardOldestPolicy()); // DiscardPolicy
List<Future> allFutures = new ArrayList<>();
// Submit tasks to the executor
for (int i = 1; i <= 5; i++) {
    final int taskNumber = i;
    executor.submit(() -> {
        System.out.println("Task " + taskNumber + " is running");
        try {
            Thread.sleep(1000); // Simulate task execution
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Task " + taskNumber + " completed");
    });
}

// Shutdown the executor
executor.shutdown();
```

Що очікуєте побачити в консолі?

Зупинка пулу потоків

service.shutdown() – ініціює процес закриття потоків.

service.isShutdown() – повертає True, якщо процес ініційовано.

service.isTerminated() – повертає True, якщо всі задачі завершені.

service.awaitTermination (10, TimeUnit.SECONDS) – блокує закриття потоків протягом визначеного часу, таким чином дає можливість задачам закінчитися.

Service.shutdownNow() – повна зупинка пулу.

```
public void destroy()
{ try {
poolExecutor.shutdown();
if (!poolExecutor.awaitTermination(AWAIT_TIMEOUT, TimeUnit.SECONDS))
{ poolExecutor.shutdownNow();
}
}
catch (InterruptedException e) { // If the current thread is interrupted, cancel all tasks again.
pool.shutdownNow(); // Maintain the interrupt status.
Thread.currentThread().interrupt();
 }
 }
```

Стани пулу

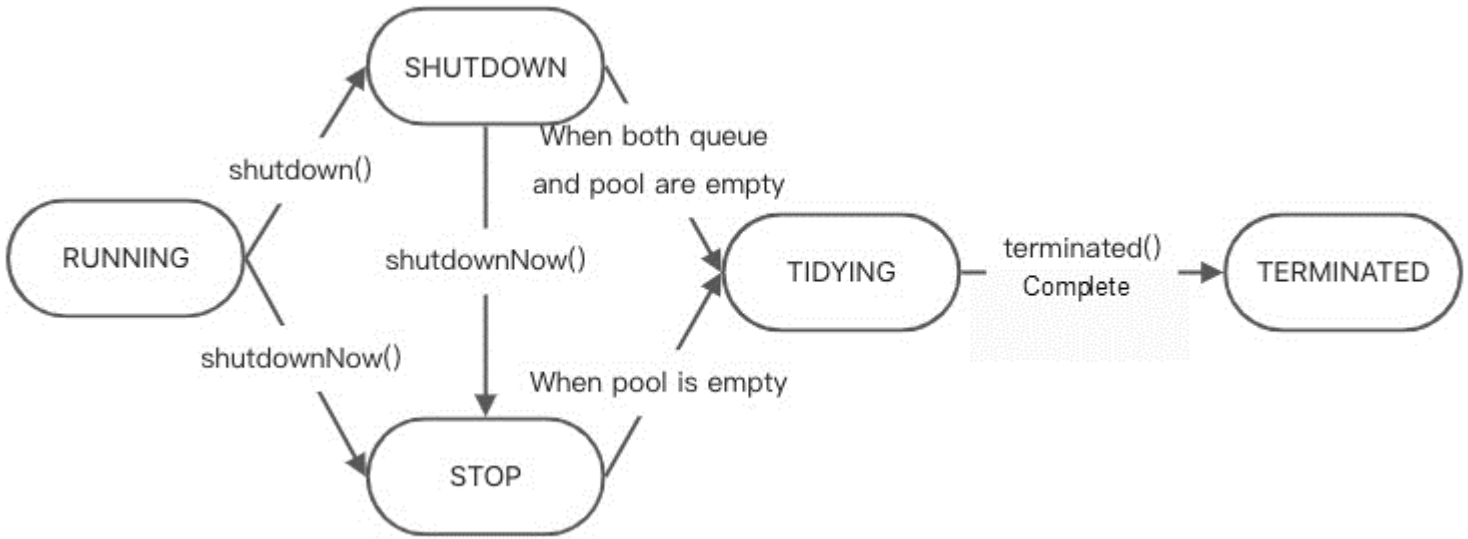
RUNNING: ThreadPoolExecutor – приймає нові задачі і виконує.

SHUTDOWN: ThreadPoolExecutor – не приймає.

STOP: ThreadPoolExecutor приймає нові задачі і не виконує, і зупиняє задачі, які виконуються.

TIDYING: ThreadPoolExecutor – в процесі зупинки.

TERMINATED: ThreadPoolExecutor пул повністю зупинено.



Питання

1. Що таке інтерфейс ExecutorService і яку роль він відіграє в керуванні виконанням задач у пулі потоків?
2. У чому різниця між методами execute() та submit() інтерфейсу ExecutorService?
3. Які можливості надають методи Future, що повертаються методом submit()?
4. Яке призначення методів shutdown() та shutdownNow() і чим вони відрізняються?
5. Як працює метод awaitTermination() і в яких випадках його доцільно використовувати?
6. Які стани може мати потік у Java та як їх можна визначити за допомогою класу Thread?
7. Яке призначення методу getState() і яку інформацію він повертає?
8. Чим відрізняються методи isAlive() та getState() при аналізі стану потоку?
9. Чому метод Thread.stop() вважається небезпечним і не рекомендований до використання?
10. Як коректно реалізувати механізм зупинки потоку за допомогою interrupt()?
11. Які переваги використання пулів потоків порівняно зі створенням потоків вручну?
12. Для яких задач доцільно використовувати FixedThreadPool і які його обмеження?
13. Чим CachedThreadPool відрізняється від FixedThreadPool з точки зору управління кількістю потоків?
14. Яке призначення ScheduledThreadPool і які типи планування задач він підтримує?
15. У яких сценаріях варто застосовувати SingleThreadExecutor і які гарантії порядку виконання він забезпечує?