Лекція 5. Атомарні змінні та асинхронні результати в багатопотокових Java-додатках

Тема 1. Класи, включені до пакета java.util.concurrent.atomic, та їх застосування для реалізації неблокуючих (lock-free) операцій синхронізації.

Тема 2. Клас DoubleAccumulator як засіб високопродуктивного накопичення числових значень у багатопотоковому середовищі.

Тема 3. Інтерфейси Future та ScheduledFuture для представлення, керування та планування асинхронних результатів обчислень.

Тема 4. Методи класу CompletableFuture для побудови, композиції та обробки асинхронних обчислень у мові Java.

WorkQueue при виконанні задач ***pop*** *використовує – набір класів які підтримують неблокуюче багатопотокове програмування на рівні* ***змінних****. Ці класи включені в пакет* java.util.concurrent.atomic.

Наприклад класи:

AtomicInteger, AtomicBoolean, AtomicLong, AtomicReference– дозволяють оновлювати та отримувати значення однієї змінної відповідного типу. Окрім конструктора, ці класи включають методи

- ✓ Get() – повертає поточне значення змінної, відповідного типу.
- ✓ set(int newValue) – змінює поточне значення змінної.
- ✓ getandset(int newValue) - повертає поточне значення змінної, після цього замінює його на нове.
- ✓ compareAndSet(int expectedValue, int newValue) – порівнює поточне значення очікуваним (expectedValue). Якщо True, тоді призначає нове значення.
- ✓ getAndIncrement() - повертає поточне значення змінної, після цього збільшує значення змінної на 1.
- ✓ getAndDecrement() - повертає поточне значення змінної, після цього зменшує значення змінної на 1.
- ✓ getAndAdd(int delta) - повертає поточне значення змінної, після цього додає до поточного значення «дельту»
- ✓ incrementAndGet()- збільшує значення змінної на 1, після цього повертає оновлене значення.
- ✓ decrementAndGet() - зменшує значення змінної на 1 , після цього повертає оновлене значення.
- ✓ addAndGet(int delta) - додає до поточного значення «дельту» після цього повертає значення.

Алгоритм без блокування (англ. Non-blocking algorithm) — підхід у паралельному програмуванні на симетрично-багатопроцесорних системах, що передбачає відмову від традиційних примітивів блокування, таких як семафори, м'ютекси і події. Розподіл доступів між потоками відбувається за допомогою атомарних операцій і спеціально розроблених під конкретну задачу механізмам синхронізації.

Перевага алгоритмів без блокування — ліпша масштабованість при збільшенні кількості процесорів. Крім того, якщо ОС перерве один з потоків фонового процесу, інші щонайменше виконають свою роботу без простою. Щонайбільше — візьмуть невиконану роботу на себе.

Алгоритми без блокувань будуються на атомарних операціях, наприклад, читання-модифікація-запис, і найбільш значуща з них — ***порівняння з обміном (CAS, Compare-and-swap).***

Приклад використання класів ***AtomicInteger, AtomicReference – в головному потоці***

```
class AtomicCounter implements Runnable {
    static AtomicInteger counter = new AtomicInteger(0);

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter.incrementAndGet();
        }
    }

    public Integer get() {
        return counter.get();
    }
}
```

```
public class Counter implements Runnable {
    private int counter = 0;

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            increment();
        }
    }

    private synchronized void increment() {
        counter++;
    }

    public synchronized int get() {
        return counter;
    }
}
```

public final int getAndAddInt(Object o, long offset, int delta)

```
public final int getAndAddInt(Object o, long offset, int
delta) {
   int v;
   do {
      v = getIntVolatile(o, offset);
   } while (!weakCompareAndSetInt(o, offset, v, v +
delta));
   return v;
}
```

Функція getAndAddInt є **низькорівневою атомарною операцією**, яка використовується у класі Unsafe в Java для виконання атомарного оновлення цілочисельного поля в пам'яті.

Вона **атомарно додає** delta до цілочисельного поля об'єкта o, яке знаходиться за зміщенням offset у пам'яті, **повертає старе значення** поля перед оновленням.

Зчитує поточне значення:

getIntVolatile(o, offset) читає значення цілочисельного поля за адресою offset у пам'яті з гарантією видимості для інших потоків (аналог volatile).

Цикл ***weakCompareAndSetInt(o, offset, v, v + delta)*** намагається атомарно оновити поле:

Якщо значення у offset все ще дорівнює v, воно змінюється на v + delta.

Якщо інший потік вже оновив значення, операція не спрацьовує, і цикл повторюється.

Цикл гарантує, що навіть при одночасних спробах оновлення значення лише один потік досягне успіху за раз.

Повертається старе значення (v) до оновлення.

Перевага:

Такий підхід **уникає блокувань**, що підвищує ефективність у багатопотоковому середовищі. Якщо значення змінюється іншим потоком до виконання CAS, цикл повторюється, гарантуючи коректність оновлення.

public final boolean compareAndSet(int expect, int update)	Перевіряє, чи поточне значення змінної дорівнює expectedValue. Якщо так, оновлює значення на newValue і повертає true. Якщо ні, нічого не змінює та повертає false.
	AtomicInteger counter = new AtomicInteger(5);
	boolean updated = counter.compareAndSet(5, 10);
	updated = counter.compareAndSet(5, 15);

public final int accumulateAndGet(int x, IntBinaryOperator accumulatorFunction)
--- 	
public final int accumulateAndGet(int x,
IntBinaryOperator accumulatorFunction) {
int prev = get(), next = 0;
for (boolean haveNext = false;;) {
if (!haveNext)
next = accumulatorFunction.applyAsInt(prev, x);
if (weakCompareAndSetVolatile(prev, next))
return next;
haveNext = (prev == (prev = get()));
}
}
--- 	
AtomicInteger acounter = new AtomicInteger(5);
boolean updated = acounter.compareAndSet(5, 10);
System.*out*.println(updated);
System.*out*.println(acounter.get());
updated = acounter.compareAndSet(5, 15);
System.*out*.println(updated);
	Гарантує безпечне оновлення змінних у багатопотоковому середовищі.
	Надає Можливість використовувати довільну функцію оновлення, наприклад:
	Додавання: accumulateAndGet(5, (a, b) -> a + b);
	Множення: accumulateAndGet(2, (a, b) -> a * b);
	Максимум: accumulateAndGet(3, Math::max);
	
	x – вхідне значення, яке буде використане в обчисленнях.
	accumulatorFunction – функція (IntBinaryOperator), яка приймає два цілі числа (поточне значення та x) і повертає нове значення.
	Використовується **нескінченний цикл**, оскільки оновлення може зазнати невдачі через паралельні зміни з боку інших потоків
	обчислюється next за допомогою переданої функції accumulatorFunction.
	атомна спроба оновлення значення від prev до next за допомогою weakCompareAndSetVolatile(prev, next).
	Якщо оновлення **успішне**, метод **повертає** next.
	Якщо інший потік змінив значення AtomicInteger, оновлення **не вдасться**.
	prev = get() отримує **актуальне** значення змінної, а haveNext перевіряє, чи залишилося воно таким самим.
	Якщо prev змінилося, цикл починається знову, і метод пробує ще раз.

public final int addAndGet(int delta)

```
@IntrinsicCandidate
public final int getAndAddInt(Object o, long offset, int delta) {
    int v;
    do {
        v = getIntVolatile(o, offset);
    } while (!weakCompareAndSetInt(o, offset, v, v + delta));
    return v;
}
```

Чим відрізняється addAndGet() від getAndAdd()?
addAndGet(int delta) спочатку додає, а потім повертає оновлене значення.

getAndAdd(int delta) спочатку повертає старе значення, а потім додає delta.
getIntVolatile(o, offset) читає поточне значення змінної (аналог volatile-зчитування); o – об'єкт, у якому знаходиться змінна; offset – зсув у пам'яті, який вказує на місце змінної всередині o.

Перевіряє, чи v (поточне значення) **не змінилося** з моменту читання.
Якщо не змінилося, то оновлює значення v + delta.
Якщо значення змінилося іншим потоком, спроба **повторюється** (do-while).

public final void lazySet(int newValue)

```
public final void lazySet(int newValue) {
    U.putIntRelease(this, VALUE, newValue);
}
```

Метод встановлює нове значення змінної **без негайної синхронізації** з іншими потоками. *U*.putIntRelease
U – це екземпляр Unsafe (внутрішній клас для роботи з пам'яттю напряму).
this – об'єкт, у якому знаходиться змінна.
VALUE – зсув змінної (отримується через Unsafe.objectFieldOffset).
newValue – нове значення, яке треба записати.

Метод lazySet корисний у багатопотокових структурах даних, де важливо зменшити витрати на синхронізацію:

Неблокуючі черги (lock-free queues) → дозволяє оновлювати покажчики head і tail без затримок.

Лічильники (counters) → зменшує навантаження на процесор при частих оновленнях.

В AtomicInteger є метод set(), який працює інакше:

set(x) → аналог volatile, тобто відразу видно іншим потокам.

lazySet(x) → запис відбудеться з невеликою затримкою, але працює швидше.

Class DoubleAccumulator

DoubleAccumulator – це клас у пакеті java.util.concurrent.atomic, призначений для ефективного атомарного обчислення значень з використанням заданої комутативної та асоціативної операції (наприклад, суми, добутку, максимуму тощо). **Дозволяє одночасні оновлення значення без блокувань**. Підходить для високонавантажених багатопотокових програм.

public DoubleAccumulator(DoubleBinaryOperator accumulatorFunction, double identity)	accumulatorFunction – бінарна операція (наприклад, Double::sum для суми). identity – початкове значення (наприклад, 0.0 для суми або 1.0 для добутку).
public void accumulate(double x)	оновлення значення
public double get()	
public void reset()	скидання до початкового значення
getThenReset()	отримати значення і скинути

accumulator.accumulate(10.5);
accumulator.accumulate(5.2);
accumulator.accumulate(3.3);
	?

maxAccumulator.accumulate(5.0);
maxAccumulator.accumulate(15.7);
maxAccumulator.accumulate(9.3);
	?

public void accumulate(double x) - реалізація

```
public void accumulate(double x) {
    Cell[] as;
    double b, v;
    int m;
    Cell a;

    if ((as = cells) != null || !casBase(b = base, b,
function.applyAsDouble(b, x))) {
        boolean uncontended = true;
        if (as == null || (m = as.length - 1) < 0 ||
            (a = as[getProbe() & m]) == null ||
            !(uncontended = a.cas(v = a.value,
function.applyAsDouble(v, x)))) {
            longAccumulate(x, function, uncontended);
        }
    }
}
```

LockFreeQueue - - алгоритм побудови lock-free черг. Перше було запропоновано - Магед М. Майклом і Майклом Л. Скоттом

Потік A починає додавати elemA:
Створює нову вершину.
Оновлює T.next → новий елемент додається в чергу.
Не встигає оновити T (воно ще вказує на попередню вершину).
Засинає через планувальник ОС.
Потік B намагається додати elemB:
Перевіряє T.next перед CAS-операцією.
T.next не null, оскільки потік A вже вставив elemA, але ще не оновив T.
CAS (T.next == null) провалюється.
Потік B застрягає в циклі повторних спроб.
Потік A прокидається:
Завершує оновлення T (переміщує T на elemA).
Тепер T.next == null, і потік B може виконати CAS.
Потік B успішно додає elemB.

Висновок:
Потік B не може прогресувати, поки потік A не оновить T.

Алгоритм має проблему гарантії прогресу, що порушує принцип lock-free.

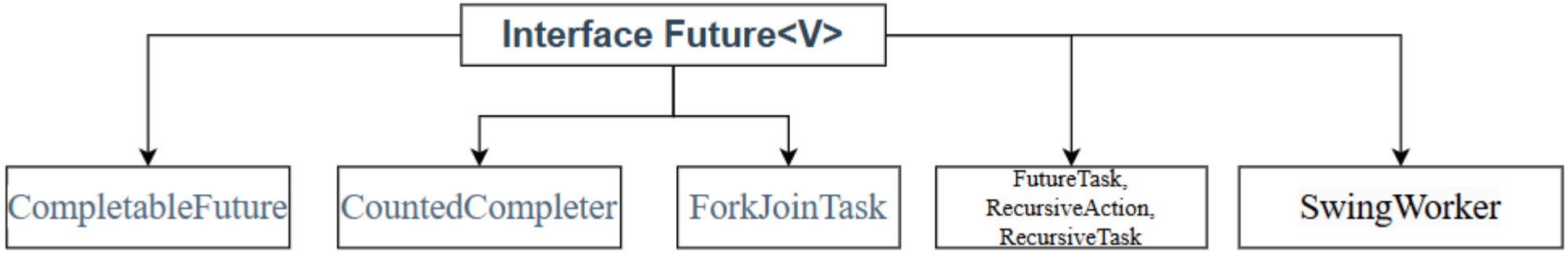
Потік B залежить від розкладу потоків у ОС.

Можливий фікс: Використання двох CAS або допоміжного маркера "pending"
для T.

```
public void enqueue(T value) {
    Node<T> newNode = new Node<>(value);
    while (true) {
        Node<T> last = tail.get();
        Node<T> next = last.next.get();

        if (last == tail.get()) {
            if (next == null) {
                if (last.next.compareAndSet(null, newNode)) {
                    tail.compareAndSet(last, newNode);
                    return;
                }
            } else {
tail.compareAndSet(last, next);
            }
        }
    }
}
```

Enqueue (enqueue):
Створює новий вузол
Знаходить хвіст
Якщо, не було ще додано елементу (next == null)
Додає елемент в хвіст черги
tail.compareAndSet(last, newNode);

В **LockFreeQueue**, якщо один потік не встигає оновити tail, інші потоки можуть це зробити самостійно.
tail.compareAndSet(last, next);




```
public static void testFuture(){
   ExecutorService executorService = Executors.newSingleThreadExecutor();
   Callable<Integer> task = () -> {
      int sum = 0;
      for (int i = 1; i <= 10; i++) {
         sum += i;
         Thread.sleep(100);
      }
      return sum;
   };
   Future<Integer> future = executorService.submit(task);
   System.out.println("Doing some other work...");

   try {

      Integer result = future.get();
      System.out.println("Result from Future: " + result);
   } catch (Exception e) {
      System.err.println("Error while retrieving the result: " + e.getMessage());
   }
   executorService.shutdown();
}
}
```

Ключові методи майбутнього:
get(): повертає результат обчислення.
Блокує, якщо обчислення не завершено.
isDone(): перевіряє, чи завершено обчислення.
cancel(): Спроба скасувати обчислення.

```
public static void getResultsnotBlocking(Future<Integer> future)
{
   if (future.isDone()) {

      Integer result = null;
      try {
         result = future.get();
      } catch (InterruptedException e) {
         throw new RuntimeException(e);
      } catch (ExecutionException e) {
         throw new RuntimeException(e);
      }
      System.out.println("Result from Future: " + result);
   } else {
      System.out.println("Still waiting for the result...");
   }
}
```

future.isDone() використовується для перевірки завершеності виконання завдання без блокування потоку. Це дозволяє уникати стану очікування і використовувати потік для інших завдань, доки асинхронне завдання виконується.

future.isDone() дозволяє періодично перевіряти завершення завдання, продовжуючи виконувати інші дії в потоці.
Якщо завдання ще виконується, програма може виконувати ("робити") в паралельному потоці.

```
public static void getResultisDone(Future<Integer> future)
{
   while (!future.isDone()) {
      System.out.println("Завдання ще виконується... ");
      try {
         Thread.sleep(500);
      } catch (InterruptedException e) {
         System.err.println("Основний потік перервано.");
      }
   }

   try {
      System.out.println("Результат: " + future.get());
   } catch (InterruptedException | ExecutionException e) {
      System.err.println("Виникла помилка.");
   }
}
```

Будемо перевіряти статус завдання кожні 0.5 sec.

```
public static void getResultbyPartBlocking(Future<Integer> future)
{
   try {
      Integer result = null; // Timeout of 500 milliseconds
      try {
         result = future.get(1000, TimeUnit.MILLISECONDS);
      } catch (InterruptedException e) {
         throw new RuntimeException(e);
      } catch (ExecutionException e) {
         throw new RuntimeException(e);
      }
      System.out.println("Result: " + result);
   } catch (TimeoutException e) {
      System.err.println("Timeout waiting for the result!");
   }
}
```

Приклади задач

Використання future.get():

Завантаження даних з сервера, де дані потрібні для обчислень.

Обчислення довготривалих значень (наприклад, пошук найкоротшого шляху).

Генерація критичних звітів (звіт не може бути пропущений).

Використання future.isDone():

Реалізація таймера або фонового логу активності, доки завдання не завершено.

Асинхронне виконання, при цьому основний потік виконує інше (наприклад, оновлення інтерфейсу).

Моніторинг стану завдань без зупинки роботи програми.

V get() throws InterruptedException, ExecutionException	За необхідності чекає завершення обчислення, а потім отримує його результат. CancellationException - завдання було відкликано ExecutionException - виняткова ситуація під час виконання завдання InterruptedException - потік було відкликано, під час очікування метод призначений для випадків, коли відомо, що завдання вже успішно завершено
boolean isCancelled()	
boolean isDone()	
boolean cancel(boolean mayInterruptIfRunning);	Може викликатись з cancel(TRUE)- метод намагається перервати завдання навіть у разі, якщо вона вже запущена та виконується. cancel(FALSE) - авдання буде скасовано лише у випадку, якщо воно ще не було розпочато. Метод повертає true, якщо завдання було успішно скасовано, і false інакше (наприклад, якщо завдання вже завершено).

public interface ScheduledFuture<V> extends Delayed, Future<V>

Відкладена результативна дія, яку можна скасувати. Зазвичай заплановане майбутнє є результатом планування завдання за допомогою ScheduledExecutorService.

Methods inherited from interface java.util.concurrent.Delayed

getDelay

Methods inherited from interface java.lang.Comparable

compareTo

Methods inherited from interface java.util.concurrent.Future

cancel, get, get, isCancelled, isDone

```
public static void testScheduleFuture() {
      ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(1);
      Runnable scheduledTask = () -> {
         int sum = 0;
         for (int i = 1; i <= 10; i++) {
            sum += i;
         }
         System.out.println(sum);
      };

      ScheduledFuture<?> scheduledFuture =
scheduledExecutorService.scheduleWithFixedDelay(
            scheduledTask, 0, 1, TimeUnit.SECONDS);

      scheduledExecutorService.shutdown();

   }
```

Метод scheduleWithFixedDelay() з ScheduledExecutorService використовується для планування повторюваних завдань із фіксованою паузою (затримкою) між завершенням одного виконання і початком наступного. Це відрізняється від scheduleAtFixedRate(), який виконує завдання з фіксованим інтервалом незалежно від часу виконання.
Ключові особливості scheduleWithFixedDelay:
Затримка між виконаннями: Завдання запускається з паузою, яка починається після завершення попереднього виконання.

Гарантоване очікування завершення попередньої задачі: Наступне виконання починається тільки після того, як попереднє завершиться (і після заданої затримки).

```
 public ScheduledFuture<?> scheduleWithFixedDelay(Runnable
command,
                                    long initialDelay,
                                    long delay,
                                    TimeUnit unit);
}
```

initialDelay – визначає початку затримку для завдання
Delay – визначає час завершенням одного виконання і початком наступного

Сценарії використання scheduleWithFixedDelay:

1. Виконання періодичних завдань із непередбаченим часом завершення

Якщо ви не знаєте, як довго триватиме завдання, і хочете забезпечити достатню паузу між повторними запусками.

Наприклад: Збирання даних із датчиків, де кожне виконання залежить від часу, необхідного для отримання чи обробки інформації.

2. Робота зі Зовнішніми ресурсами (API / Системами) - Запити до API через регулярні інтервали часу.

Наприклад: Ви відправляєте дані на сервер, і вам потрібна затримка перед наступною операцією для обробки попереднього запиту.

3. Фонові задачі, які потребують очищення чи моніторингу

Моніторинг ресурсів, таких як перевірка статусу бази даних, файлової системи, або статусу зовнішніх сервісів.

Завдання, яке очищає старі дані або тимчасові файли раз на кілька хвилин.

4. Завдання з навантаженням

Якщо завдання споживає великий обсяг ресурсів (процесор, мережа), і ви хочете, щоб вони виконувались із паузами між завершенням чергового виконання.

5. Логічний таймер для нерегулярної роботи

Наприклад: Завдання автоматичного виконання, яке не потребує повного контролю часу початку, а лише фіксовану паузу перед повторним запуском.

```
public static void logging()
{
   ScheduledExecutorService executorService = Executors.newScheduledThreadPool(1);
   Runnable task = () -> {
      System.out.println("Запис логу: " + System.currentTimeMillis());
   };
   executorService.scheduleWithFixedDelay(task, 0, 5, TimeUnit.SECONDS);
}
```

public interface RunnableScheduledFuture<V> extends RunnableFuture<V>, ScheduledFuture<V>

RunnableScheduledFuture є інтерфейсом у Java Concurrency API, який використовується для планування повторюваних або відкладених завдань. Цей інтерфейс поєднує можливості RunnableFuture (щоб завдання можна було виконувати в потоці) та ScheduledFuture (для роботи з відкладеними або періодичними завданнями).

```
public ScheduledFuture<?> schedule(Runnable
command,
                                   long delay,
                                   TimeUnit unit) {
    if (command == null || unit == null)
        throw new NullPointerException();
    RunnableScheduledFuture<Void> t =
decorateTask(command,
        new ScheduledFutureTask<Void>(command, null,
                                      triggerTime(delay, unit),
                                      sequencer.getAndIncrement()));
    delayedExecute(t);
    return t;
}
```

ScheduledFutureTask: Основний механізм, який управляє завданням і реалізує RunnableScheduledFuture. Він об'єднує функції Runnable, Future і ScheduledFuture.
triggerTime(): Метод обчислює час запуску завдання, додаючи затримку delay до поточного часу.
Цей метод конвертує затримку delay у базові одиниці часу (зазвичай у наносекунди) і додає її до поточного часу системи (System.nanoTime()). Таким чином, він визначає момент у майбутньому, коли завдання має бути виконане.

sequencer.getAndIncrement(): Унікальний номер (sequence number), який використовується для ідентифікації черговості завдань.
decorateTask(): Метод, який можна перевизначити для створення власних версій завдань. Стандартно він повертає передане завдання без змін.

Modifier and Type	Method and Description
boolean	**isPeriodic**() Returns `true` if this task is periodic.

Methods inherited from interface java.util.concurrent.RunnableFuture

run

Methods inherited from interface java.util.concurrent.Delayed

getDelay

Methods inherited from interface java.lang.Comparable

compareTo

Methods inherited from interface java.util.concurrent.Future

cancel, get, get, isCancelled, isDone

```
public class RunnablePeriodic
{
   static class CustomScheduledThreadPoolExecutor extends
ScheduledThreadPoolExecutor {
      public CustomScheduledThreadPoolExecutor(int corePoolSize) {
         super(corePoolSize);
      }

      @Override
      protected <V> RunnableScheduledFuture<V> decorateTask(Runnable
runnable, RunnableScheduledFuture<V> task) {
         // Повертаємо наше завдання для подальшого аналізу
         return super.decorateTask(runnable, task);
      }
   }
}
```

```
public static void testRunnable(){
   RunnablePeriodic.CustomScheduledThreadPoolExecutor
executor = new
RunnablePeriodic.CustomScheduledThreadPoolExecutor(1);

   Runnable oneTimeTask = () ->
System.out.println("[Одноразове завдання] Виконано!");

   Runnable periodicTask = () ->
System.out.println("[Періодичне завдання] Виконується...");

   ScheduledFuture<?> oneTimeFuture =
executor.schedule(oneTimeTask, 2, TimeUnit.SECONDS);

   ScheduledFuture<?> periodicFuture =
executor.scheduleAtFixedRate(periodicTask, 0, 3,
TimeUnit.SECONDS);

   System.out.println("Одноразове завдання isPeriodic(): " +
         ((RunnableScheduledFuture<?>)
oneTimeFuture).isPeriodic()); // Очікуємо false

   System.out.println("Періодичне завдання isPeriodic(): " +
         ((RunnableScheduledFuture<?>)
periodicFuture).isPeriodic()); // Очікуємо true

   executor.shutdown();
}
```

public class CompletableFuture<T> extends Object implements Future<T>, CompletionStage<T>

CompletableFuture — це: Клас, який реалізує інтерфейс Future, розширюючи його функціональність.

- Асинхронне обчислення:

Завдання запускаються в окремих потоках.
Основні методи для створення асинхронних завдань: supplyAsync() і runAsync().

- Чейнінг операцій:

Можливість виконувати кілька пов'язаних операцій, які залежать одна від одної:
Використовуються методи thenApply(), thenAccept(), thenRun().

- Обробка результату:

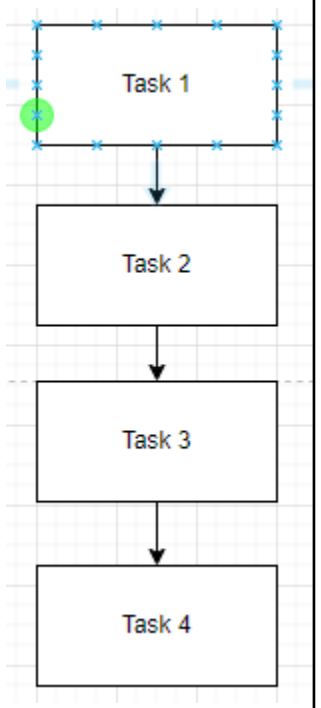
Ви можете обробити результат після завершення завдання, а також отримати результат асинхронного виконання.

- Обробка винятків:

Можливість легко обробляти винятки за допомогою методів exceptionally() і handle().
Паралельне обчислення:
Можливість запускати кілька задач одночасно і комбінувати їх результати через allOf() або anyOf().

Метод	Опис
`supplyAsync(Supplier<T>)`	Запускає асинхронне завдання, яке повертає результат.
`runAsync(Runnable)`	Запускає асинхронне завдання без повернення результату.

Розглянемо програму, в якій потік має виконувати задачі 1-4 точно в порядку, які ми визначили, тобто після задачі 1 -> задача 2 -> задачі 3-> задача 4
Для цього використовують CompletableFuture



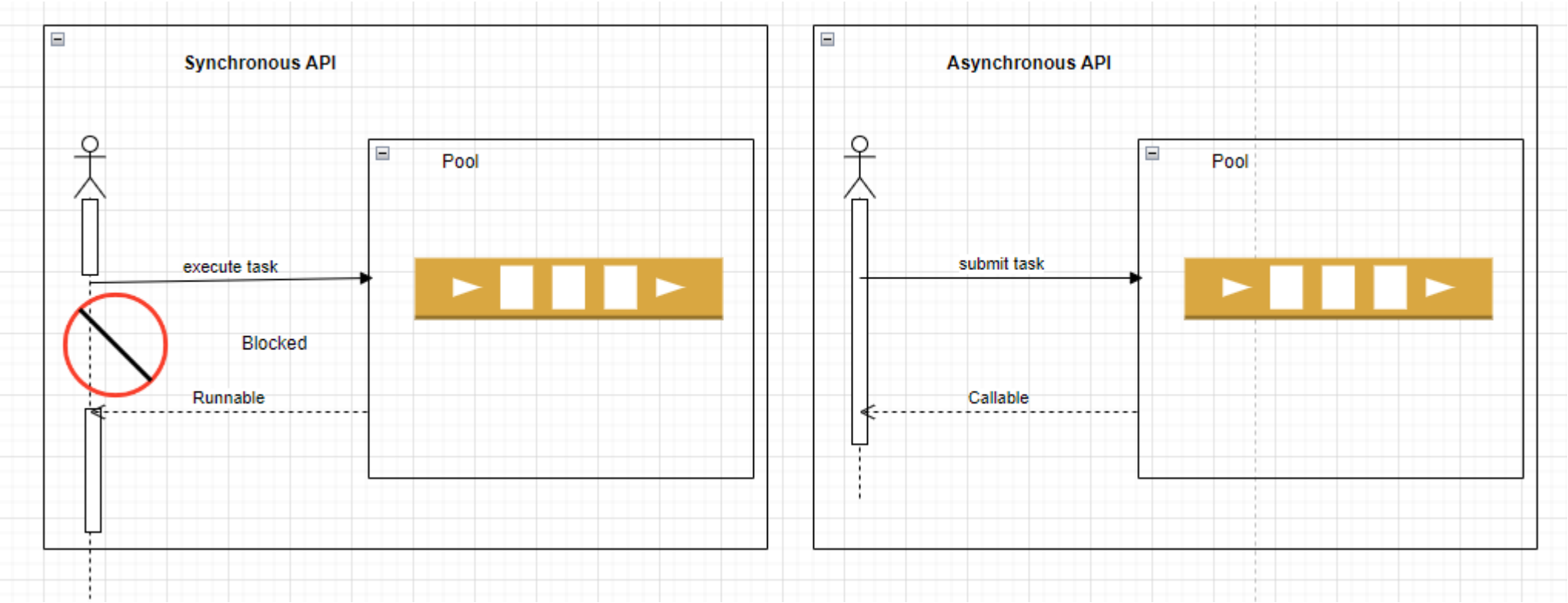

```
private static void executetaskinorder() {
    CompletableFuture<Void> future = CompletableFuture
            .runAsync(() -> task1())
            .thenRun(() -> task2())
            .thenRun(() -> task3());


    try {
        future.get(); // Blocking call to wait for completion of all
tasks
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
private static void task1() {
    System.out.println("Task 1 completed");
}
private static void task2() {
    System.out.println("Task 2 completed");
}
private static void task3() {
    System.out.println("Task 3 completed");
}
```

Метод CompletableFuture.runAsync() – включає задачі, які мають бути виконані спочатку, тобто задачі ініціалізації.
Метод CompletableFuture.thenAsync() - включає задачі, які мають бути виконані після завершення ініціалізації.

Метод	Опис
`thenApply(Function<T, R>)`	Виконує операцію над отриманим результатом і повертає новий результат.
`thenAccept(Consumer<T>)`	Виконує дію над отриманим результатом без повернення значення.
`thenRun(Runnable)`	Виконує асинхронну дію після завершення поточного завдання, не бере результат.
`thenCompose(Function<T, CompletableFuture<R>>)`	Компонує залежні асинхронні завдання, повертає новий `CompletableFuture`.

Class CompletableFuture<T>

CompletableFuture використовується для ***асинхронного*** програмування в Java. Асинхронність — це процес обробки введення/виводу, що дозволяє продовжити обробку інших завдань, не чекаючи завершення попереднього завдання. Таким чином, основний потік не блокується і не передбачає завершення завдання, а значить, може паралельно виконувати й інші завдання. Такий паралелізм значно підвищує продуктивність програми.



CompletableFuture — це розширення Future API

Future використовується як посилання на результат асинхронної задачі. У ньому є метод isDone() для перевірки, завершилася чи задача чи ні, а також метод get() для отримання результату після його завершення.

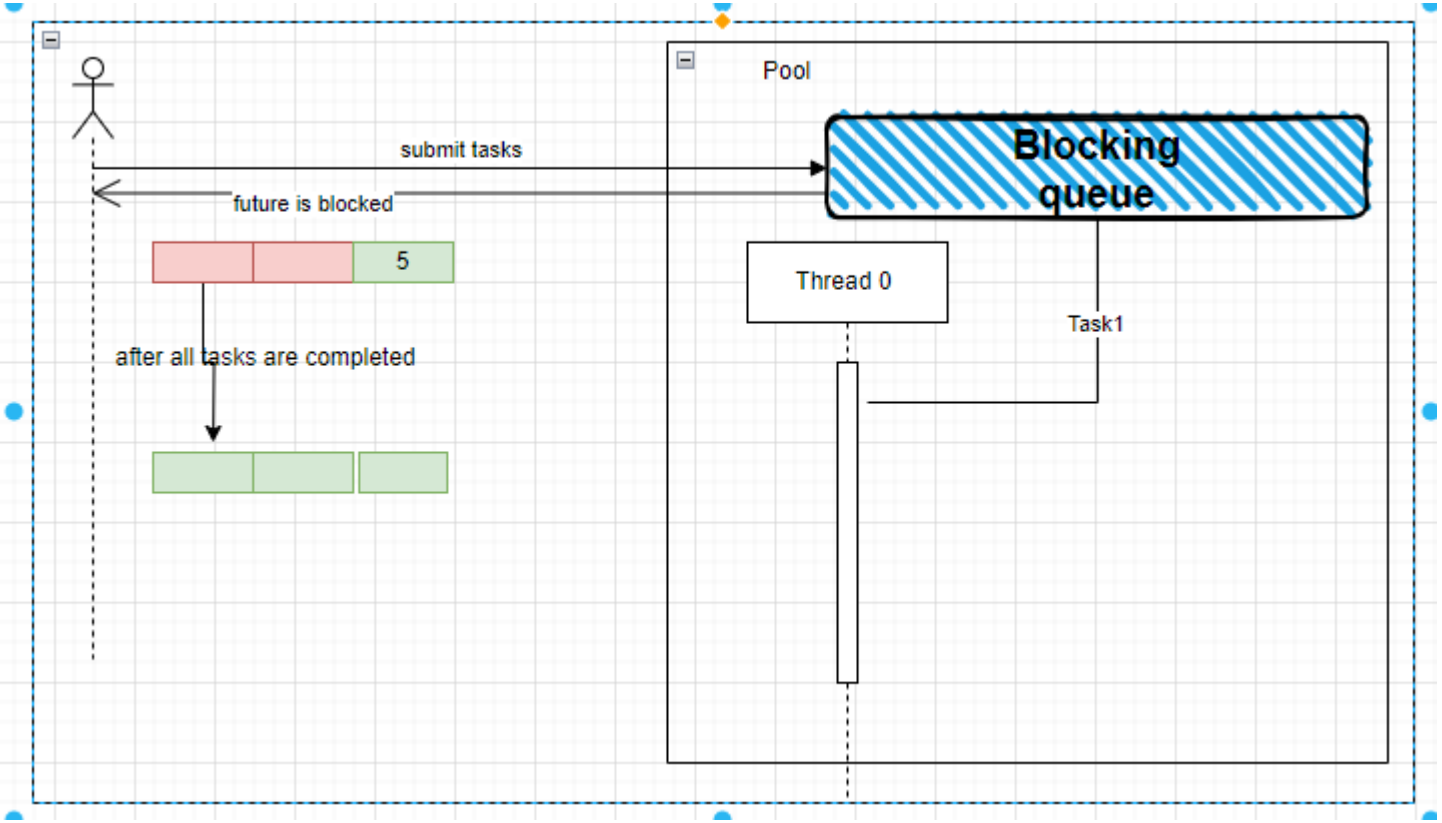
Обмеження Future API

1. Future чекає відповіді від потоку, який виконує задачу, якщо цей потік з яких причин не відповідає, тоді Future «зависає»

```
ExecutorService service = Executors.newFixedThreadPool(1);
Future <Integer> future = service.submit(new CallableTask());
try {
   //якщо задачу не закінчено то потік буде заблоковано
  Integer result=  future.get();
  System.out.println("Result from the task is "+ result+"  ");
} catch (InterruptedException e) {
   throw new RuntimeException(e);
} catch (ExecutionException e) {
   throw new RuntimeException(e);
}
service.shutdown();
```

Обмеження Future 2. Організація відповідей методу call для більш ніж однієї задачі

1) Задачі буду закінчуватись в «неочікуваному» порядку. Для алгоритмів, які передбачають порядок виконання задач -методи Future не можуть вирішити проблему.



Питання

1. Яке призначення класу CompletableFuture і які переваги він надає порівняно з інтерфейсом Future?
2. Які методи CompletableFuture використовуються для запуску асинхронних задач (runAsync, supplyAsync)?
3. У чому полягає різниця між методами thenApply, thenAccept та thenRun?
4. Як працюють методи композиції thenCompose та thenCombine і в яких випадках їх доцільно використовувати?
5. Які можливості обробки помилок надає клас CompletableFuture (exceptionally, handle, whenComplete)?
6. Яке призначення класу ReentrantLock і чим він відрізняється від механізму synchronized?
7. Які основні методи класу ReentrantLock забезпечують керування блокуванням (lock, tryLock, lockInterruptibly, unlock)?
8. Що таке справедливе (fair) блокування в ReentrantLock і як воно впливає на продуктивність?
9. У чому полягає перевага використання ReentrantReadWriteLock у багатопотокових сервісах із переважанням операцій читання?
10. Як використання окремих блокувань на читання та запис (readLock, writeLock) впливає на масштабованість і продуктивність кешів?