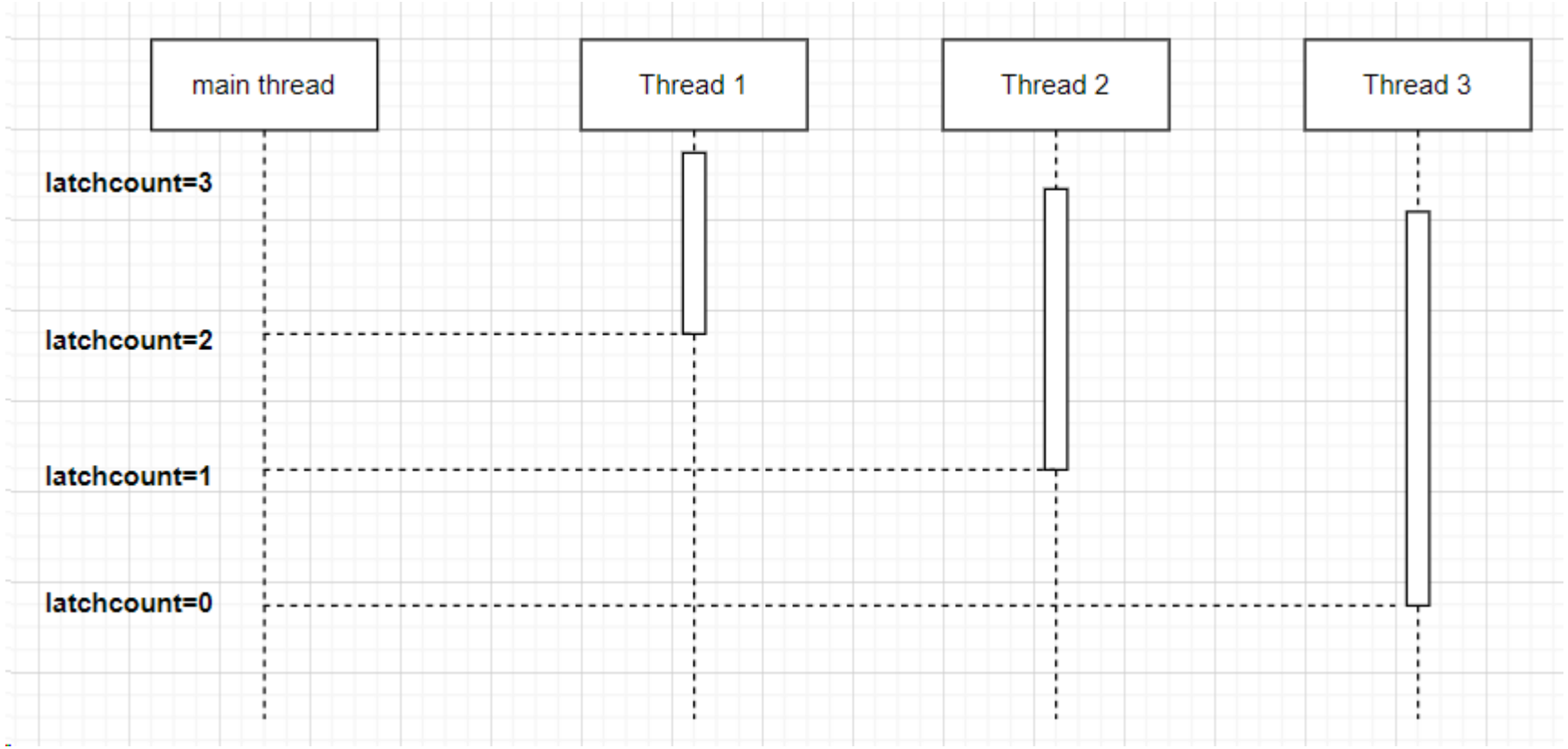
Лекція 7. Удосконалені алгоритми та механізми блокування

Тема 1. Блокування за допомогою лічильника з можливістю зупинки блокування. Клас CountDownLatch.

Тема 2. Бар’єрна синхронізація з можливістю перевикористання створеного бар’єру. Клас CyclicBarrier.

Тема 3. Бар’єрна синхронізація з можливістю перевикористання бар’єру та динамічного додавання сторін. Клас Phaser.

Class CountDownLatch - методи класу дозволяють призупинити виконання потоки, доки певні залежні потоки не будуть виконані. При створенні екземпляру класу CountDownLatch визначається кількість залежних потоків, і очікування триває доки зворотній відлік, який здійснюється методом countdown() для заданого числа не буде дорівнювати 0. CountDownLatch — це універсальний інструмент синхронізації, який можна використовувати для багатьох цілей. Розглянемо типовий приклад, Маємо головний потік, який не може виконувати свої задачі без попередньої ініціалізації (наприклад, визначення зв'язку з базу даних, і т.п.), тобто головний потік залежить від 3х інших потоків. Для організації такої залежності, використовуємо CountDownLatch, ініціалізовану як N, щоб змусити один потік чекати, доки N потоків виконає певну дію або якась дія буде виконана N разів.



```
import java.util.concurrent.CountDownLatch;
public class CountDownLatchExample {
   public static void main(String[] args) throws InterruptedException {
CountDownLatch latch = new CountDownLatch(3);
new Thread(new Task(latch), "Thread1").start();
      new Thread(new Task(latch), "Thread2").start();
      new Thread(new Task(latch), "Thread3").start();
latch.await();
System.out.println("All threads have finished execution!");
   }
   static class Task implements Runnable {
      private CountDownLatch latch;

      Task(CountDownLatch latch) {
         this.latch = latch;
      }

      @Override
      public void run() {
         System.out.println(Thread.currentThread().getName() + " is running");
         try {
            Thread.sleep(2000);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
         System.out.println(Thread.currentThread().getName() + " has finished");
         latch.countDown();
}
   }
}
```

Створюємо екземпляру класу CountDownLatch з параметром 3. Створюємо клас, який реалізує інтерфейс Runnable, в конструкт класу передаємо екземпляр CountDownLatch. В методі run() – зменшуємо лічильник на 1 викликом методу CountDownLatch.

В головному потоці – стартуємо залежні потоки і призупиняємо головний потік викликом latch.await().

В результаті, головний потік буде «пробуджено», тоді, коли виконається останній залежний потік.

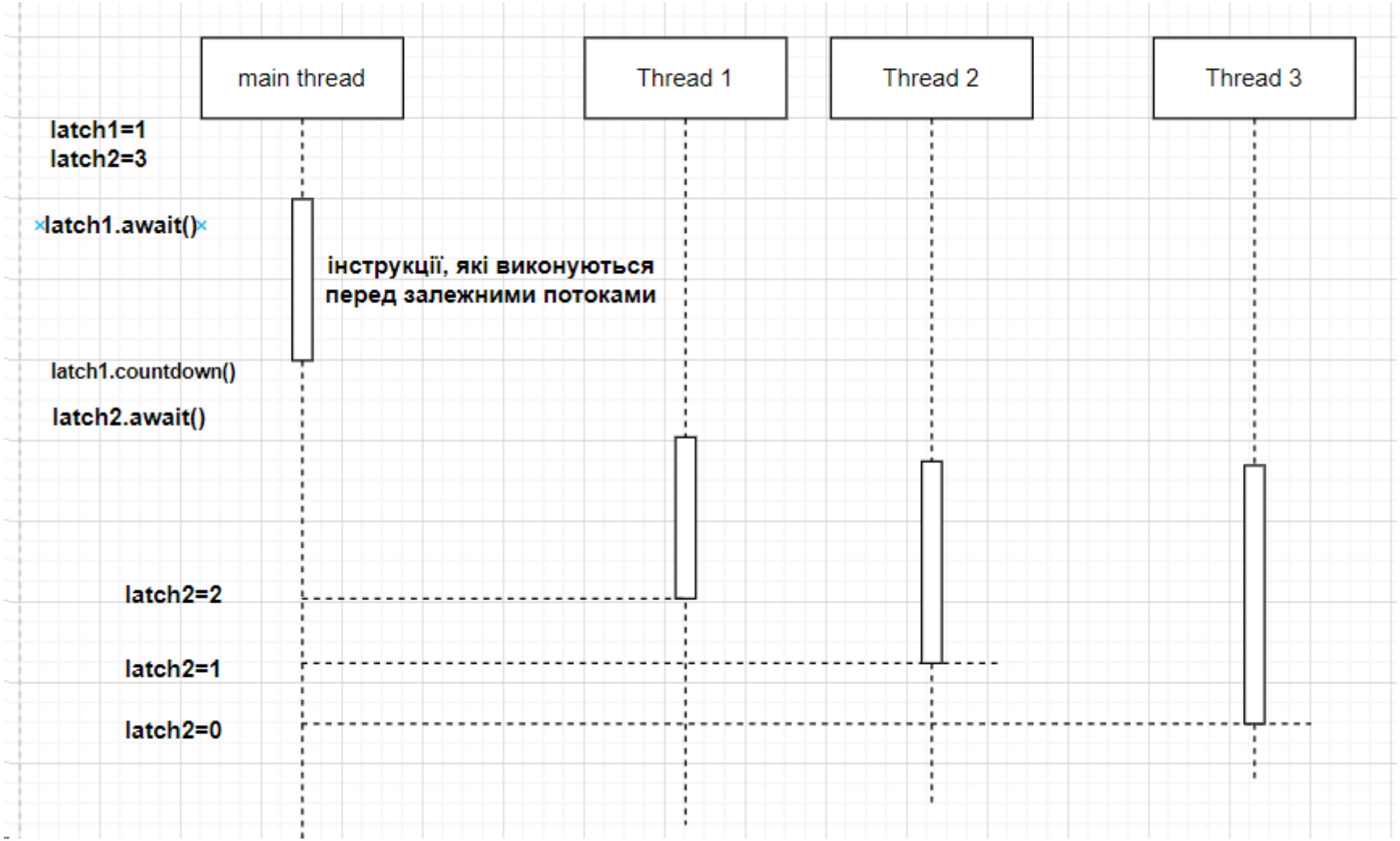
CountDownLatch, ініціалізований лічильником один, служить простою засувкою ввімкнення/вимкнення або шлюзом: усі викликані потоки очікують на шлюзі, поки він не буде відкритий потоком, що викликає countDown().

```
import java.util.concurrent.CountDownLatch;

public class Main {

    public static void main(String[] args) {
        CountDownLatch latch1 = new CountDownLatch(1);
        CountDownLatch latch2 = new CountDownLatch(3);

for (int i = 0; i < 3; i++) {
            final int worker = i;
            new Thread(() -> {

                try {
                    latch1.await();
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }

                System.out.println("Worker thread " + worker + " has started.");

                try {
                    // Simulate time taken for task with sleep
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }

                System.out.println("Worker thread " + worker + " has completed.");
                    latch2.countDown();
            }).start();
        }
        try {
            System.out.println("Main thread preparing worker threads...");
            Thread.sleep(3000);
            System.out.println("Main thread has opened latch1 gate! Worker threads can proceed.");
         latch1.countDown();
        latch2.await();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        System.out.println("All worker threads have completed their tasks, main thread can now proceed...");
    }
}
```

latch1.await() – блокує старт залежних потоків, доки не будуть виконані певні інструкції. Виклик методу latch1.countDown() – визнає кінець блокування для залежних потоків.

latch2.await() – блокує головний потік, доки не виконаються залежні потоки, завершення яких визначиться коли latch2.countDown() дорівнює 0.

CountDownLatch, ініціалізований лічильником один, служить простою засувкою ввімкнення/вимкнення



Методи класу CountDownLatch

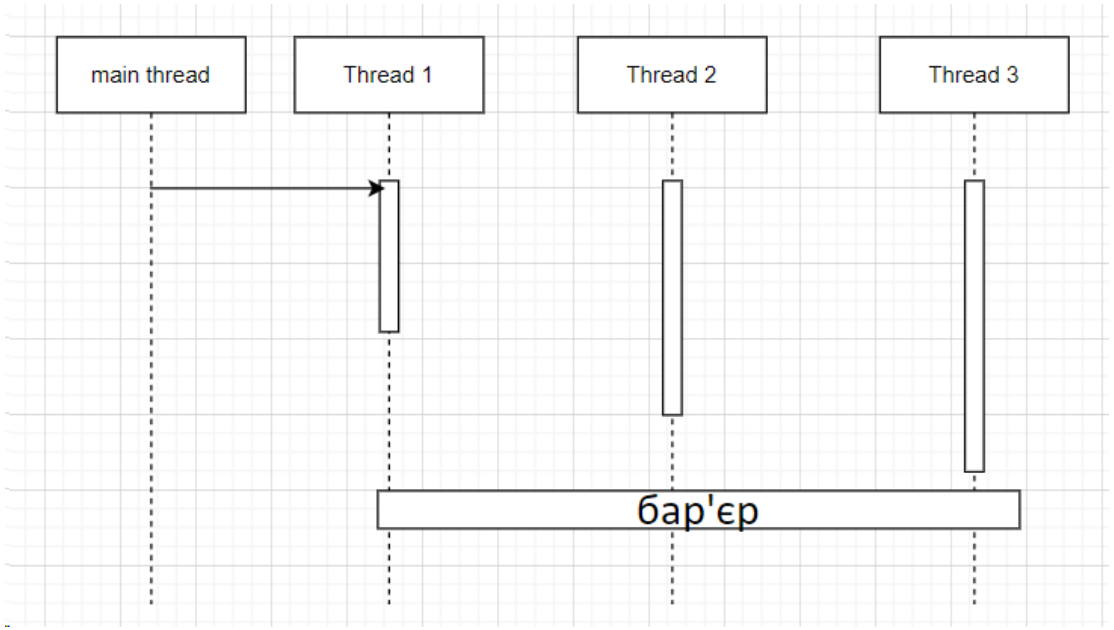
public CountDownLatch(int count)	Конструктор класу, визначає кількість потоків, які мають буди виконані перед «розблокуванням» головного потоку.
public void await() throws InterruptedException	Блокує поточний потік (Thread.Current) доки лічильник не дорівнює 0. Якщо призупинений потік буде відкликано викликом «interrupt» підчас його блокування, тоді await() – генерує виняток InterruptedException
public boolean await(long timeout, TimeUnit unit) throws InterruptedException	Блокує поточний потік (Thread.Current) доки лічильник не дорівнює 0 або доки не витрачено час на очікування. Якщо призупинений потік буде відкликано викликом «interrupt» підчас його блокування, тоді await() – генерує виняток InterruptedException.
public void countDown()	Зменшує лічильник на 1.
public long getCount()	Повертає поточне значення лічильника

Від класу Object наслідує методи: clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Недолік класу `CountDownLatch` - це одноразовий бар'єр синхронізації. Коли фіксатор (лічильник) досягає нуля, його неможливо скинути або змінити, і всі потоки, що очікують, звільняються. Іншу поведінку отримаємо методами класу CyclicBarrier.

CyclicBarrier - це точка синхронізації, де визначена кількість потоків зустрічається та очікує один одного. Коли всі досягнуть спільної точки бар’єру, сторони звільняються, і бар’єр можна використовувати повторно.

Наприклад, головний потік має задачу обчислити загальний результат, як суму результатів залежних потоків. Приймаючи до уваги, що час завершення обчислення буде відрізнятись для кожного потоку – їх треба призупинити доки не має жодного потоку, який не завершив задачу. Інструкція «призупинення» носить назву «бар'єру».



```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
public class CyclicBarrierExample {
    public static void main(String[] args) {
        final CyclicBarrier barrier = new CyclicBarrier(3,
new Runnable() {
            @Override
            public void run() {
                System.out.println("All tasks are completed.
Let's sum up!");
                System.out.println("Sum = " +
(Computation1.result + Computation2.result +
Computation3.result));
            }
        });
System.out.println("Number of parties required to trip
the barrier = " + barrier.getParties());
        Thread t1 = new Thread(new
Computation1(barrier));
        Thread t2 = new Thread(new
Computation2(barrier));
        Thread t3 = new Thread(new
Computation3(barrier));
        t1.start();
        t2.start();
        t3.start();
        System.out.println("Main thread has finished");
    }
}
```

```
static class Computation1 implements Runnable
{
        public static int result = 0;
        private CyclicBarrier barrier;

        public Computation1(CyclicBarrier barrier)
{
            this.barrier = barrier;
        }
        @Override
        public void run() {
            result = 2 * 2;
            try {
                barrier.await();
            } catch (InterruptedException |
BrokenBarrierException e) {
                e.printStackTrace();
            }
        }
    }
```

```
 static class Computation2 implements
Runnable {
        public static int result = 0;
        private CyclicBarrier barrier;

        public Computation2(CyclicBarrier
barrier) {
            this.barrier = barrier;
        }

        @Override
        public void run() {
            result = 3 + 3;
            try {
                barrier.await();
            } catch (InterruptedException |
BrokenBarrierException e) {
                e.printStackTrace();
            }
        }
    }
```

```
static class Computation3
implements Runnable {
        public static int result =
0;
        private CyclicBarrier
barrier;

        public
Computation3(CyclicBarrier
barrier) {
            this.barrier = barrier;
        }

        @Override
        public void run() {
            result = 4 / 2; // e.g.
this is another computation
task
            try {
                barrier.await();
            } catch
(InterruptedException |
BrokenBarrierException e) {
                e.printStackTrace();
            }
        }
    }
}
```

barrier.await() – змушує потік чекати, доки всі 3 потоки не виконають задачу обчислення. Метод barrier.getParties() – повертає кількість потоків, які очикуються.

Термін «Cyclic» в назві класу CyclicBarrier – означає, що один раз створений екземпляр класу, можна використовувати для визначення багатьох бар'єрів. Визначений бар’єр може бути скасовано в результаті виклику методу reset(). Скасування «бар’єру» може бути виконано в наступних сценаріях:

- ✓ Зламаний бар’єр: якщо один із потоків, який очікує інші переривається або його метод await() закінчується, тоді CyclicBarrier перейде в непрацюючий стан. Це необхідно, щоб інші потоки, що очікують, не потрапили в тупик. У цьому сценарії, для повторного використання екземпляру CyclicBarrier, знадобиться скинути «існуючий» «барєр» за допомогою методу reset().

- ✓ Ранній перезапуск: коли відомо, що жоден з потоків не досягне бар’єру, і тоді потрібно перезапустити весь процес. Це може статися, наприклад, якщо виникла проблема з однією або декількома "сторонами", так що ви знаєте, що вони ніколи не досягнуть бар'єру.

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CycleBarrierExample {

    public static void main(String[] args) {

        CyclicBarrier barrier = new CyclicBarrier(3);

        new Worker(barrier, "Thread 1").start();
        new Worker(barrier, "Thread 2").start();
        new Worker(barrier, "Thread 3").start();

        // Now, these three workers can use the barrier again
        barrier.reset();

        new Worker(barrier, "Thread 4").start();
        new Worker(barrier, "Thread 5").start();
        new Worker(barrier, "Thread 6").start();
    }
}

class Worker extends Thread {
    private CyclicBarrier barrier;

    public Worker(CyclicBarrier barrier, String name) {
        super(name);
        this.barrier = barrier;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " Waiting on barrier");
        try {
            barrier.await();
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " Crossing the barrier");
    }
}
```

В даному прикладі, 3 потоки - "Thread 1", "Thread 2", "Thread 3", пов'язані одним «бар'єром», який скасовано після «їх» старту.
Після цього, з цим же «бар'єром» визначені потоки "Thread 1", "Thread 2", "Thread 3».

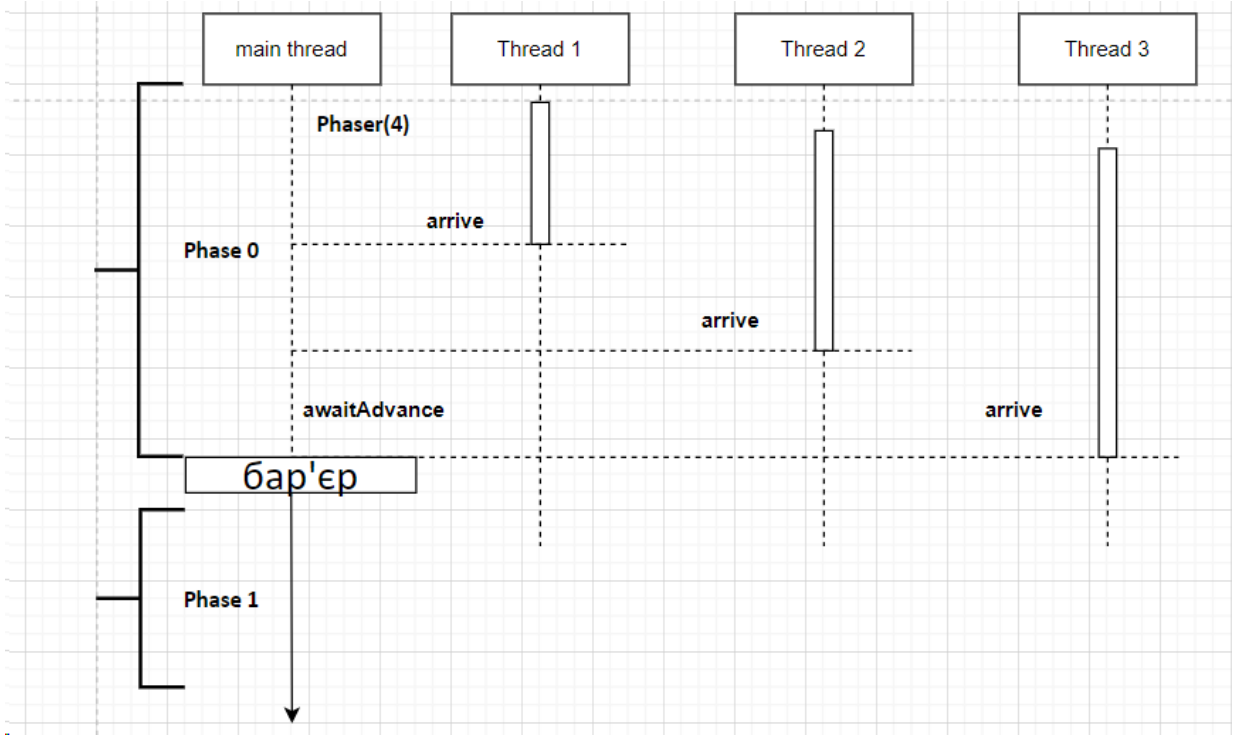
При виконанні програми можливий наступний результат

```
Thread 1 Waiting on barrier
Thread 2 Waiting on barrier
Thread 3 Waiting on barrier
Thread 4 Waiting on barrier
Thread 1 Crossing the barrier
Thread 2 Crossing the barrier
Thread 3 Crossing the barrier
Thread 4 Crossing the barrier
Thread 5 Waiting on barrier
Thread 6 Waiting on barrier

java.util.concurrent.BrokenBarrierException
                at java.base/java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:252)
                at java.base/java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:364)
                at Worker.run(CycleBarrierExample.java:35)
```

Методи класу **CyclicBarrier**

public CyclicBarrier(int parties,Runnable barrierAction)	Конструктор класу, визначає кількість потоків, які мають буди зупинені «бар'єром». Задачу, яку треба виконати, після того, як всі потоки досягли «бар’єру».
public int getParties()	Повертає кількість потоків, які мають буди зупинені «бар'єром»
public int await()throws InterruptedException, BrokenBarrierException	«Зупиняє» потік (Thread.Current) після того, як задача виконана і якщо він не є останнім потоком який очікують. Якщо призупинений потік буде відкликано викликом «interrupt» підчас його блокування, тоді await() – генерує виняток InterruptedException. Якщо «бар’єр» буде скасовано, тоді генерується виняток BrokenBarrierException
public boolean isBroken()	Повертає TRUE, якщо «бар’єр» зруйновано, через те що один з потоків, було відкликано методом interrupt
public void reset()	Повертає «бар’єр» у початковий стан, для потоків які очікую, буде згенеровано виняток BrokenBarrierException.
public int getNumberWaiting()	Повертає кількість потоків, які очікують інших перед «бар'єром»

Class Phaser – реалізує поведінку CyclicBarrier and CountDownLatch, при цьому є більш гнучким і може синхронізувати різні фази обчислень і підходить для більш складних сценаріїв синхронізації. Його також можна багаторазово використовувати, подібно до `CyclicBarrier` при цьому кількість потоків(сторін) можна динамічно змінювати. Конструктор Phaser – визначає кількість потоків, включно з головним потоком. Кожен потік, після закінчення задачі визначається, як «arrive», головний потік чекає на «прибуття» всі потоків. Кожен проміжок часу від ініціалізації екземпляру Phaser до прибуття всіх потоків визначається номером фази, починаючи з нуля.



```
import java.util.concurrent.Phaser;

class PhaserTask implements Runnable {

   private Phaser phaser;

   PhaserTask(Phaser phaser) {
      this.phaser = phaser;
   }

   @Override
   public void run() {
      System.out.println(Thread.currentThread().getName() + " has arrived and is working in Phase-" +
phaser.getPhase());
      phaser.arriveAndAwaitAdvance();

      System.out.println(Thread.currentThread().getName() + " has finished working on Phase-" +
phaser.getPhase());
   }
}
public class PhaserExample {
   public static void main(String[] args) {
      Phaser phaser = new Phaser(3); // Main thread, Thread-0, Thread-1
      new Thread(new PhaserTask(phaser), "Thread-0").start();
      new Thread(new PhaserTask(phaser), "Thread-1").start();

      System.out.println(Thread.currentThread().getName() + " has arrived and is working in Phase-" +
phaser.getPhase());
      phaser.arriveAndAwaitAdvance();
      System.out.println(Thread.currentThread().getName() + " has finished working on Phase-" +
phaser.getPhase());
   }
}
```

В наведеному прикладі, Phaser ініціалізується трьома сторонами: основним потоком, потоком-0 і потоком-1.

Клас PhaserTask створює завдання, яке буде виконуватися кожним потоком. Метод run() спочатку повідомляє про надходження до Phaser, виконує певну роботу (відображає повідомлення), а потім чекає на надходження інших потоків.

Основний метод запускає два інші потоки (Thread-0 і Thread-1), а потім повідомляє про своє надходження до Phaser. Він виконує деяку роботу (відображає повідомлення), а потім чекає, поки прийдуть інші потоки.

Коли всі потоки прибули до Phaser, усі вони продовжують роботу та виконують наступний набір завдань.

Ця концепція особливо корисна в програмах, які включають серію кроків, які повинні виконуватися декількома потоками в «lockstep», де всі вони повинні завершити крок 0, перш ніж будь-якому з них буде дозволено перейти до кроку 1 і так далі.

Динамічна зміна кількості потоків при синхронізація методами класу Phaser

```
import java.util.concurrent.Phaser;

public class PhaserExample {

    public static void main(String[] args) {
        Phaser phaser = new Phaser(1);
        System.out.println("Phase " + phaser.getPhase() + " has started...");
        for (int i = 0; i < 3; i++) {
            int finalI = i;
            Thread thread = new Thread(() -> {
                phaser.register();
                System.out.println(finalI + " is performing task for Phase " + phaser.getPhase());
                phaser.arriveAndAwaitAdvance();
                System.out.println(finalI + " is performing task for Phase " + phaser.getPhase());
                phaser.arriveAndAwaitAdvance();
                phaser.arriveAndDeregister();
            });
            thread.start();
        }
        phaser.arriveAndAwaitAdvance();

        System.out.println("Phase 0 has finished...");
        System.out.println("Phase " + phaser.getPhase() + " has started...");
        for (int i = 0; i < 2; i++) {
            int finalI = i;
            Thread thread = new Thread(() -> {
                phaser.register();
                System.out.println(finalI + " is performing task for Phase " + phaser.getPhase());
                phaser.arriveAndAwaitAdvance();
                phaser.arriveAndDeregister();
            });
            thread.start();
        }
        phaser.arriveAndAwaitAdvance();
        System.out.println("Phase 1 has finished..."); }}
```

В цьому прикладі для кожної фази реєструємо потоки, коли вони починаються, і скасовуємо їх, коли вони закінчуються. Коли сторони завершують своє виконання в кожній фазі, вони скасовують реєстрацію в Phaser, таким чином змінюючи кількість сторін у наступній фазі.

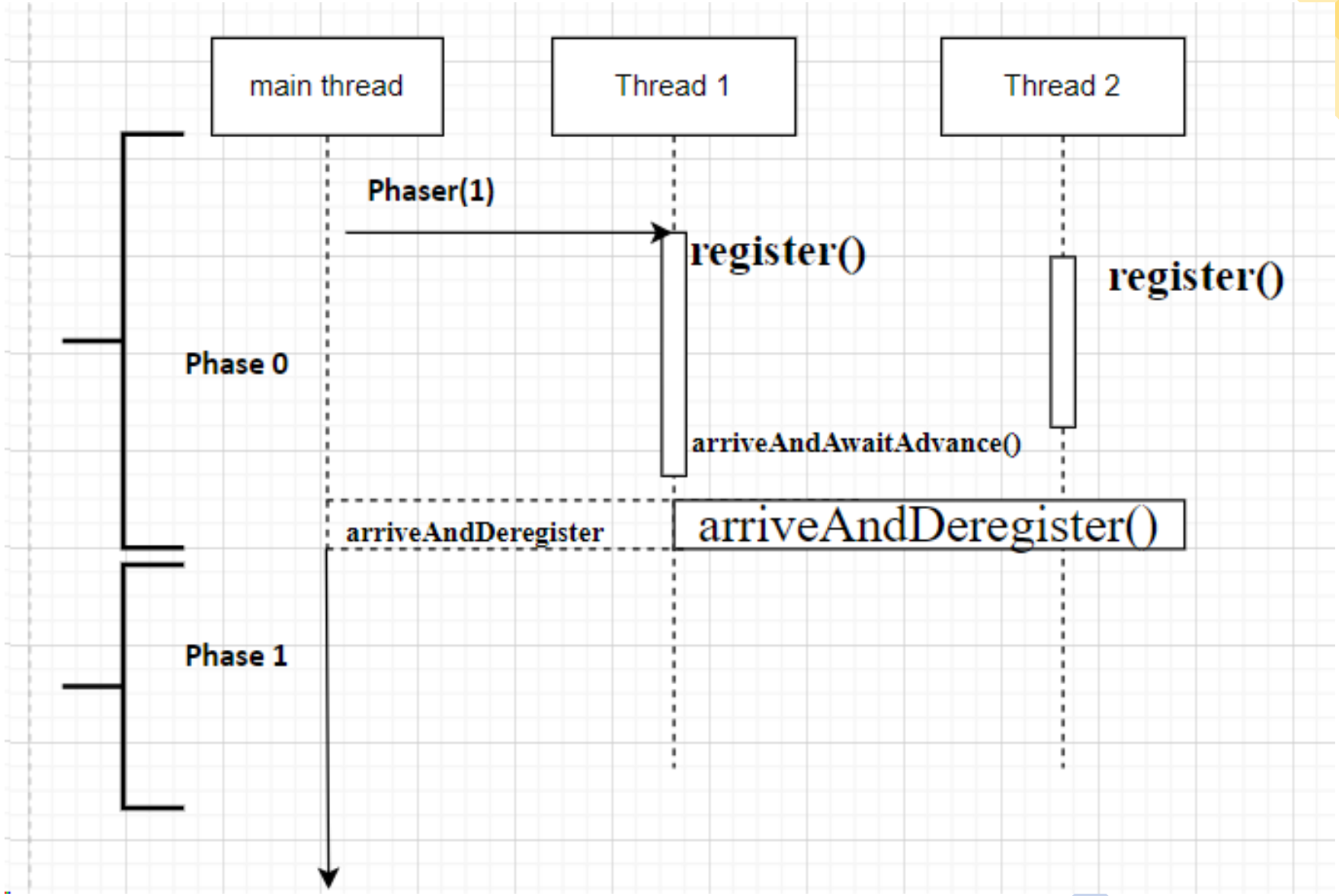
Фаза 0 - Phaser ініціалізується тільки основним потоком.

phaser.register() – додає потік до фази 0 при його старті.

phaser.arriveAndAwaitAdvance() –потік чекає на «прибуття» всі потоків.

phaser.arriveAndDeregister() – скасовує реєстрацію доданого потоку, як тільки інші потоки прибули.

За необхідністю, циклічно починаємо наступну фазу.


main thread
Thread 1
Thread 2
Phaser(1)
register()
register()
Phase 0
arriveAndAwaitAdvance()
arriveAndDeregister
arriveAndDeregister()
Phase 1

Відмінність між Phaser.arrive() та Phaser.arriveAndAwaitAdvance()

```
final Phaser phaser = new Phaser(3);

new Thread(() -> {

    System.out.println("Thread 1 completed the task and waiting for others.");
    phaser.arriveAndAwaitAdvance(); System.out.println("All threads completed. Thread 1 continues...");
}).start();

new Thread(() -> {

    System.out.println("Thread 2 completed the task and moving on.");
    phaser.arrive();
}).start();

phaser.arriveAndAwaitAdvance();
System.out.println("Main thread continues after all threads have arrived.");
```

Phaser.arrive() використовується, коли потік завершив виконання, і йому не потрібно чекати інших. Потік сповіщає про своє прибуття і продовжує своє виконання. Phaser.arrive() використовується, коли не потрібна синхронізація потоків в деякій загальній точці (як у CyclicBarrier), але є потреба відстежувати завершення фази. Phaser.arrive() не блокує потік виклику.

Phaser.arriveAndAwaitAdvance() використовується, коли потрібно створити точку синхронізації, де потоки виконують різні частини завдання та чекають, поки прийдуть інші потоки. Усі потоки, що викликають цей метод, будуть заблоковані, доки не прибудуть усі сторони.

У цьому прикладі потоки 1 і основний потік викликають arriveAndAwaitAdvance() і чекатимуть на цьому етапі, доки всі 3 сторони не викличуть primiAndAwaitAdvance.
Потік 2 не чекає інших і продовжує своє виконання після виклику arrive(). Основний потік продовжується лише після завершення потоку 1 і потоку 2, по суті створюючи точку синхронізації.

Методи класу Phaser

public Phaser()	Конструктор класу без параметрів, створює екземпляр, який можна використовувати тільки разом з методом register(), тобто додавати потоки динамічно
public Phaser(int parties)	Конструктор класу з параметром parties, створює екземпляр, з визначеною кількістю потоків
public Phaser(Phaser parent, int parties)	Конструктор класу з параметром «батьківський» Phaser - створює новий Phaser у дереві фазової синхронізації. Така ініціалізація необхідна, для обробки складних сценаріїв синхронізації шляхом створення деревоподібної структури об’єктів Phaser, де, як правило, кожен вузол (об’єкт Phaser) у дереві представляє фазу обчислення та має батьківський Phaser, який синхронізує інші пов’язані фази.
public int register()	Виклик методу додає потік до групи потоків, які будуть очікуватися для завершення Фази. Метод повертає номер Фази, до якої було зареєстровано потік. Якщо повернене значення від’ємне, це означає, що ця фаза закінчилася, і в цьому випадку реєстрація не має ефекту.
public int bulkRegister(int parties)	Виклик методу додає задану кількість потоків, які будуть очікуватися для завершення Фази. Метод повертає номер Фази, до якої було зареєстровано потік. Якщо повернене значення від’ємне, це означає, що ця фаза закінчилася, і в цьому випадку реєстрація не має ефекту.

Методи класу Phaser

public int arrive()	Повідомлення про прибуття потоку, без очікування на інші потоки. Метод повертає номер Фази, на якій потік прибув. Якщо повернене значення від'ємне, тоді потік прибув після завершення фази.
public int arriveAndAwaitAdvance()	Повідомлення про прибуття потоку з очікування на інші потоки. Метод повертає номер Фази, на якій потік прибув. Якщо повернене значення від'ємне, тоді потік прибув після завершення фази.
public int arriveAndDeregister()	Повідомлення про прибуття потоку і одночасно зняття з реєстрації в групі Фази. Скасування реєстрації зменшує кількість сторін, необхідних для наступної Фази. Якщо Фаза має батьківську фазу, тоді скасування реєстрації призводить до того, що цей потік також скасовано з батьківської Фази.
public int awaitAdvance(int phase)	Визначає фазу, яку має чекати головний потік
public final int getPhase()	Повертає номер поточної фази.
public int getRegisteredParties()	Повертає кількість потоків (сторін), які зареєструвались в поточній фазі.
public int getArrivedParties()	Повертає кількість зареєстрованих потоків (сторін), які досягли поточної фази.
public int getUnarrivedParties()	Повертає кількість зареєстрованих потоків (сторін), які не досягли поточної фази.

Методи класу Phaser

public Phaser getParent()	Повертає «батьківський» екземпляру типу Phaser для поточного екземпляру типу Phaser
public Phaser getRoot()	Повертає «корінь» для поточного екземпляру типу Phaser
public boolean isTerminated()	Повертає TRUE, якщо екземпляру типу Phaser було скасовано.

Питання

1. Яке призначення класу CountDownLatch і яку задачу синхронізації він розв’язує?
2. Як працює лічильник у CountDownLatch і яким чином він впливає на блокування потоків?
3. Які основні методи надає клас CountDownLatch (await, countDown) і як вони використовуються?
4. Чи можна повторно використовувати об’єкт CountDownLatch після досягнення лічильником нуля і чому?
5. У яких сценаріях CountDownLatch є більш доцільним за join() або wait/notify?
6. Що таке бар’єрна синхронізація і яке призначення класу CyclicBarrier?
7. У чому полягає можливість перевикористання (cyclic) бар’єру в CyclicBarrier?
8. Яке призначення бар’єрної дії (barrier action) та коли вона виконується?
9. Що відбувається, якщо один із потоків не досягає бар’єру або переривається?
10. У яких випадках доцільно використовувати CyclicBarrier у паралельних алгоритмах?
11. Яке призначення класу Phaser і чим він концептуально відрізняється від CyclicBarrier?
12. Як Phaser підтримує динамічне додавання та видалення «сторін» (participants)?
13. Що таке фаза (phase) в Phaser і як відбувається перехід між фазами?
14. Які основні методи класу Phaser використовуються для керування синхронізацією (arrive, arriveAndAwaitAdvance, register)?
15. У яких сценаріях Phaser є більш гнучким та ефективним рішенням порівняно з CountDownLatch і CyclicBarrier?