СИНХРОНІЗАТОРИ З JAVA CONCURRENCY UTILITIES

Питання 6.4.

Синхронізатори

- Найбільш поширеними є 4 типи синхронізаторів:
  - Замки зі зворотним відліком (Countdown latch),
  - Циклічні бар’єри та етапники
  - Обмінники (exchangers),
  - Семафори

Countdown latch

- Замок з відліком змушує одну або декілька ниток чекати у воріт (gate), доки інша нитка не відкриє ці ворота та дозволить виконуватись чекаючим ниткам далі.
  - Він складається з лічильника (count) та операцій, що «змушують нитку чекати, поки лічильник не досягне нуля» та «декрементують лічильник».

- Клас java.util.concurrent.CountDownLatch реалізує синхронізатор типу countdown latch.
  - Екземпляр CountDownLatch ініціалізується конкретним count, викликаючи конструктор CountDownLatch(int count).
  - Конструктор може викидати IllegalArgumentException, коли count від’ємний.

Методи класу CountDownLatch

- **void await()**: змушує викликаючий потік очікувати, доки летч не відрахує до 0, якщо потік в цей час не перерветься (тоді виникне InterruptedException).
- **boolean await(long timeout, TimeUnit unit)**: змушує викликаючий потік очікувати, доки летч не відрахує до 0 або заданого timeout-значення. Повертає true, коли count досягає 0 або false, коли час очікування минув. Викидання InterruptedException аналогічне.
- **void countDown()**: декрементує count, звільняючи всі очікуючі потоки, коли count досягне 0. Якщо count уже 0, нічого при виклику не станеться.
- **long getCount()**: повертає поточний count. Корисний для тестування та відладки.
- **String toString()**: повертає рядок, що ідентифікує даний летч, і його стан. Стан, in brackets, включає рядковий літерал "Count =", за яким йде поточний count.

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

public class CountDownLatchDemo
{
   final static int NTHREADS = 3;

   public static void main(String[] args)
   {
      final CountDownLatch startSignal = new CountDownLatch(1);
      final CountDownLatch doneSignal = new CountDownLatch(NTHREADS);
      Runnable r = new Runnable()
                   {
                      @Override
                      public void run()
                      {
                         try
                         {
                            report("entered run()");
                            startSignal.await(); // wait until told to proceed
                            report("doing work");
                            Thread.sleep((int) (Math.random() * 1000));
                            doneSignal.countDown(); // reduce count on which
                                                    // main thread is waiting
                         }
                         catch (InterruptedException ie)
                         {
                            System.err.println(ie);
                         }
                      }

                      void report(String s)
                      {
                         System.out.println(System.currentTimeMillis() + ": " +
                                            Thread.currentThread() + ": " + s);
                      }
                   };
```

- Часто countdown latch використовується для перевірки того, що нитки почали роботу приблизно одночасно.
- Головна нитка спочатку створює пару countdown latch.
  - 1) startSignal – перешкоджає виконанню будь-якої робочої нитки (worker thread), поки головна нитка (main thread) не буде готова до їх виконання.
  - 2) doneSignal – змушує головну нитку чекати, поки виконання всіх робочих ниток не закінчиться.

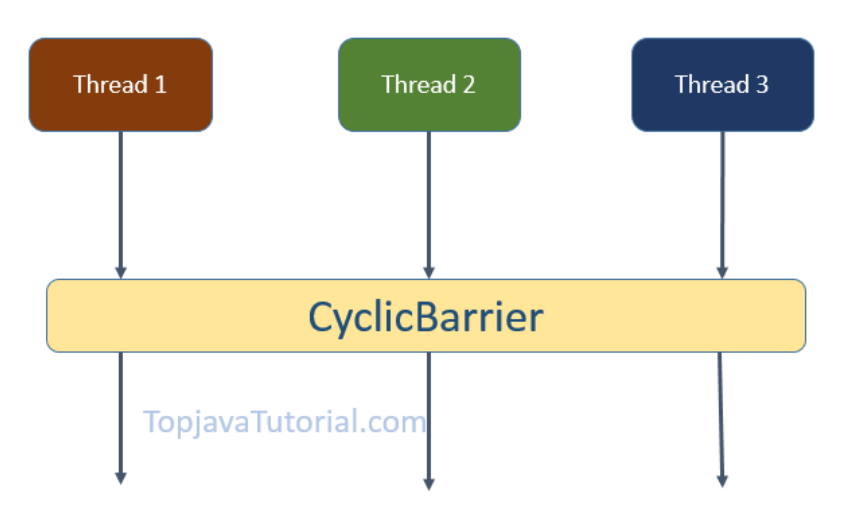
```
    ExecutorService executor = Executors.newFixedThreadPool(NTHREADS);
    for (int i = 0; i < NTHREADS; i++)
       executor.execute(r);
    try
    {
       System.out.println("main thread doing something");
       Thread.sleep(1000); // sleep for 1 second
       startSignal.countDown(); // let all threads proceed
       System.out.println("main thread doing something else");
       doneSignal.await(); // wait for all threads to finish
       executor.shutdownNow();
    }
    catch (InterruptedException ie)
    {
       System.err.println(ie);
    }
  }
}
```

- Потім головна нитка створює runnable-об’єкт, чий метод run() виконується послідовно створеними робочими потоками.
  - Спочатку цей метод виводить початкове повідомлення, а потім викликає метод startSignal.await() для очікування моменту, коли лічильник цього замка зі зворотним відліком зчитає 0 before it can proceed.
  - Як тільки це сталось, run() виводить повідомлення, яке індикує виконану роботу та засинає на випадковий період часу (від 0 до 999 мс) для симуляції роботи.
  - На цей момент, run() викликає метод countDown() doneSignal для декрементації лічильника летча.
  - Як тільки лічильник доходить до 0, головний потік, який очікував на цей сигнал, продовжить роботу, вимикаючи екзекутор та перериваючи роботу додатку.

- Після створення runnable головний потік отримує екзекутор, який базується на пулі з NTHREADS потоків, а далі викликає метод execute() екзекутора NTHREADS разів, передаючи runnable кожному з NTHREADS потоків пулу.
  - Ця дія запускає робочий потік, який входить у run().
- Потім головний потік
  - виводить повідомлення та засинає на 1с для симуляції виконання додаткової роботи (дає шанс усім робочим потокам зайти в run() та викликати startSignal.await()),
  - викликає метод countDown() startSignal, щоб змусити робочі потоки запуститись,
  - Виводить повідомлення, що indicate that it's doing something else,
  - Викликає метод await() doneSignal, щоб почекати на досягнення нуля лічильником countdown latch, before it can proceed.

```
main thread doing something
1384281251694: Thread[pool-1-thread-1,5,main]: entered run()
1384281251694: Thread[pool-1-thread-2,5,main]: entered run()
1384281251694: Thread[pool-1-thread-3,5,main]: entered run()
main thread doing something else
1384281252723: Thread[pool-1-thread-3,5,main]: doing work
1384281252723: Thread[pool-1-thread-2,5,main]: doing work
1384281252723: Thread[pool-1-thread-1,5,main]: doing work
```

Клас java.util.concurrent.CyclicBarrier реалізує даний синхронізатор



- Екземпляр класу CyclicBarrier ініціалізується конкретним числом parties (потоки працюють на спільну ціль) за допомогою конструктора CyclicBarrier(int parties).
  - Конструктор викидає IllegalArgumentException, коли передається значення parties < 1.
- Альтернатива: CyclicBarrier(int parties, Runnable barrierAction), додається параметр barrierAction, який виконується після спрацювання бар'єру.
  - Іншими словами, коли parties-1 потоків очікують, а ще один прибуває, цей потік виконує barrierAction, а потім всі потоки продовжують роботу.
  - Цей runnable корисний для оновлення загальної статистики до того, як будь-який потік продовжить роботу.
  - Конструктор викидає IllegalArgumentException, коли передане значення parties менше за 1.
  - Попередній конструктор викликає цей конструктор, передаючи null замість barrierAction; не буде виконуватись runnable-задача, коли бар'єр is tripped.)

Методи класу CyclicBarrier

- ***int await()***: змушує викликаючий потік чекати, поки всі частини викличуть await() циклічного бар'єра.
  - Викликаючий потік припиняє чекати, коли він чи інший чекаючий потік переривається, у іншого потоку закінчується час в процесі або інший потік викликає reset() для циклічного бар'єру.
  - Якщо викликаючий потік має статус «перерваний» на вході чи переривається в процесі очікування, метод викидає InterruptedException.
  - Викидає java.util.concurrent.BrokenBarrierException, коли для бар'єру викликано reset(), поки будь-який потік очікує або коли бар'єр порушено через виклик await().
  - Коли будь-який потік переривається в процесі очікування, всі інші очікуючі потоки викидають BrokenBarrierException, а бар'єр переходить у порушений (broken) стан
  - Якщо викликаючий потік є останнім з потоків, що надходять, а не-null значення barrierActoin було передано в конструкторі, викликаючий потік виконує цю runnable-задачу до дозволу продовжуватись іншим потокам.
  - Цей метод повертає індекс прибуття викликаючого потоку, де індекс getParties()-1 вказує на перший потік, що прибув, а 0 – на останнього прибувшого.

Методи класу CyclicBarrier

- ***int await(long timeout, TimeUnit unit)***: еквівалентний попередньому методу, за винятком можливості визначення тривалості можливого очікування викликаючого потоку. Викидає виняток TimeoutException, коли відведений час завершується, поки потік очікує.
- ***int getNumberWaiting()***: повертає кількість parties, які в цей момент очікують на бар'єрі. Метод корисний для налагодження разом з твердженнями.
- ***int getParties()***: повертає кількість parties, необхідних для подолання бар'єру.
- ***boolean isBroken()***: повертає true, коли одна або кілька parties розбивають бар'єр у зв'язку з переривання м або таймаутом з моменту конструювання циклічного бар'єру, останнього перезапуску або коли barrier action failed через виняток. Інакше повертає false.
- ***void reset()***: перезапускає бар'єр в початковий стан. Якщо всі parties на даний момент в стані очікування на бар'єрі, вони повернуть BrokenBarrierException. Зауважте, що якщо перезапуски трапились після прориву з інших причин, їх буде складно обробити. Потоки потрібно пересинхронізувати по-іншому та обрати потік для перезапуску. Тому можлива потреба у створенні нового бар'єру для подальшого використання.

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

public class CyclicBarrierDemo
{
   public static void main(String[] args)
   {
      float[][] matrix = new float[3][3];
      int counter = 0;
      for (int row = 0; row < matrix.length; row++)
         for (int col = 0; col < matrix[0].length; col++)
         matrix[row][col] = counter++;
      dump(matrix);
      System.out.println();
      Solver solver = new Solver(matrix);
      System.out.println();
      dump(matrix);
   }
```

```
    static void dump(float[][] matrix)
    {
       for (int row = 0; row < matrix.length; row++)
       {
          for (int col = 0; col < matrix[0].length; col++)
             System.out.print(matrix[row][col] + " ");
          System.out.println();
       }
    }
 }
```

- Циклічні бар’єри корисні у сценаріях паралельної декомпозиції, де тривалі задачі розбиваються на підзадачі, індивідуальні результати яких потім поєднуються (merge) в загальний результат.
  - Головний потік спочатку створює квадратну матрицю дробових чисел і записує її в стандартний потік виводу (output stream).
  - Потім цей потік інстанціює клас Solver, який створює окремий потік для виконання обчислень з кожним рядком.
  - Далі змінена матриця виводиться (dumped).

```
class Solver
{
   final int N;
   final float[][] data;
   final CyclicBarrier barrier;

   class Worker implements Runnable
   {
      int myRow;
      boolean done = false;

      Worker(int row)
      {
         myRow = row;
      }

      boolean done()
      {
         return done;
      }

      void processRow(int myRow)
      {
         System.out.println("Processing row: " + myRow);
         for (int i = 0; i < N; i++)
            data[myRow][i] *= 10;
         done = true;
      }
```

```
      @Override
      public void run()
      {
         while (!done())
         {
            processRow(myRow);

            try
            {
               barrier.await();
            }
```

```
            catch (InterruptedException ex)
            {
               return;
            }
            catch (BrokenBarrierException ex)
            {
               return;
            }
         }
      }
   }
```

- Solver представляє конструктор, який отримує аргумент (matrix) і зберігає посилання на неї у полі data разом з кількістю рядків N.
  - Далі конструктор створює циклічний бар’єр з N parties та barrier action, яка відповідає за злиття (merging) всіх рядків у фінальну матрицю.
  - Наприкінці конструктор створює робочий потік, який виконує окремий Worker runnable, який відповідає за обробку окремого рядка матриці.
- Конструктор чекає, поки робочі потоки не закінчать роботу.
  - метод run() робочого потоку періодично викликає processRow() для конкретного рядка, поки done() повертає true (у прикладі - після одноразового виконання processRow()).
  - Після повернення processRow(), яке вказує на те, що рядок було оброблено, робочий потік викликає await() для циклічного бар’єру; він не може продовжувати роботу.

```
public Solver(float[][] matrix)
{
   data = matrix;
   N = matrix.length;
   barrier = new CyclicBarrier(N,
                               new Runnable()
                               {
                                  @Override
                                  public void run()
                                  {
                                     mergeRows();
                                  }
                               });
   for (int i = 0; i < N; ++i)
      new Thread(new Worker(i)).start();

   waitUntilDone();
}
```

```
void mergeRows()
{
   System.out.println("merging");
   synchronized("abc")
   {
      "abc".notify();
   }
}

void waitUntilDone()
{
   synchronized("abc")
   {
      try
      {
         System.out.println("main thread waiting");
         "abc".wait();
         System.out.println("main thread notified");
      }
      catch (InterruptedException ie)
      {
          System.out.println("main thread interrupted");
      }
   }
}
}
```

- У певний момент всі робочі потоки вже викличуть await().
  - Коли останній потік (обробляє останній рядок у матриці) викликає await(), відбудеться перемикання (trigger) бар'єрної дії, що поєднує всі оброблені рядки в результуючу матрицю.
  - У даному прикладі merger не вимагається, проте він потрібний для більш складних випадків.
- Фінальна задача, що виконується в mergeRows(), сповіщає головний потік, викликаний конструктором Solver.
  - Потік очікує на монітор, пов'язаний з String-об'єктом "abc".
  - Виклик notify() здатний «розбудити» очікуючий потік, який є єдиним потоком, що чекає на монітор.

```
0.0 1.0 2.0
3.0 4.0 5.0
6.0 7.0 8.0

main thread waiting
Processing row: 1
Processing row: 2
Processing row: 0
merging
main thread notified

0.0 10.0 20.0
30.0 40.0 50.0
60.0 70.0 80.0
```

Етапники (Phasers) - більш гнучка реалізація циклічних бар’єрів

- На відміну від циклічного бар’єра, який керує фіксованою кількістю потоків, етапник може керувати їх змінною кількістю і реєструвати їх в будь-який момент.
  - Щоб реалізувати цю можливість, етапник (phaser) використовує етапи та phase numbers.
  - Етап (phase) – поточний стан етапника, який визначається цілочисельною кількістю етапів (phase number).
  - Коли останні зареєстровані потоки підходять до бар’єру, етапник просувається до наступного етапу та інкрементує кількість етапів (phase number) на 1.
- Клас java.util.concurrent.Phaser реалізує етапник. Основні конструктори та методи:
  - Phaser(int threads) – конструктор, який створює етапник, який координує nthreads потоків (які ще не підійшли до бар’єру) з кількістю етапів (phase number) = 0.
  - int register() – метод, який додає новий потік, що ще не прибув до бар’єру, до етапника та повертає phase number, щоб класифікувати прибуття. Це число відоме як arrival phase number.
  - int arriveAndAwaitAdvance() – метод, що записує прибуття та очікує на просування етапника (яке стається після того, як інші потоки прибули). Повертає phase number, до якого прибуття відбувається.
  - int arriveAndDeregister() - метод arrives at this phaser and deregisters from it, не очікуючи на прибуття решти, зменшуючи кількість потоків, що потрібні для просування на майбутніх етапах.

```
public class PhaserDemo
{
   public static void main(String[] args)
   {
      List<Runnable> tasks = new ArrayList<>();
      tasks.add(() -> System.out.printf("%s running at %d%n",
      tasks.add(() -> System.out.printf("%s running at %d%n",
                                        Thread.currentThread().getName(),
                                        System.currentTimeMillis()));
      runTasks(tasks);
   }
   static void runTasks(List<Runnable> tasks)
   {
      final Phaser phaser = new Phaser(1); // "1" (register self)
      // create and start threads
      for (final Runnable task: tasks)
      {
         phaser.register();
         Runnable r = () ->
                      {
                         try
                         {
                            Thread.sleep(50 + (int) (Math.random() * 300));
                         }
                         catch (InterruptedException ie)
                         {
                            System.out.println("interrupted thread");
                         }
                         phaser.arriveAndAwaitAdvance(); // await the ...
                                                         // creation of ...
                                                         // all tasks
                         task.run();
                      };
         Executors.newSingleThreadExecutor().execute(r);
      }
      // allow threads to start and deregister self
      phaser.arriveAndDeregister();
   }
}
```

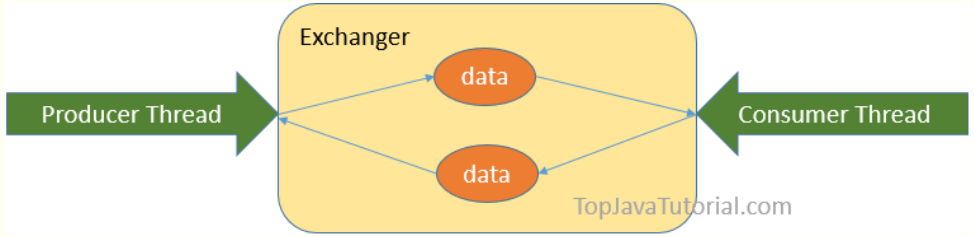
Використання етапника для управління One-Shot Action Serving a Variable Number of Parties

- Головний потік за замовчуванням створює пару runnable-задач, кожна з яких повідомляє час (в мс) свого запуску.
- Потім ці задачі запускаються після створення екземпляру Phaser, який очікує прибуття обох задач до бар’єру.

Вивід програми

- Буде подібний вивід (додаток має не закінчувати роботу – натисніть Ctrl+C чи символьний еквівалент завершення додатку):
  - pool-1-thread-1 running at 1445806012709
  - pool-2-thread-1 running at 1445806012712
- Як очікується, обидва потоки починають працювати одночасно (в цьому випадку), навіть незважаючи на можливе відкладення на 349 мс через наявність Thread.sleep().
- Закоментуйте phaser.arriveAndAwaitAdvance(); // await the ...
- та помітите, що потоки запускаються в абсолютно різні моменти:
  - pool-2-thread-1 running at 1445806212870
  - pool-1-thread-1 running at 1445806213013

Обмінники (Exchangers)



- Обмінник забезпечує точку синхронізації, в якій нитки можуть обмінюватись об’єктами.
  - Кожна нитка представляє деякий об’єкт на вході в метод exchange() обмінника, matches with a partner thread, та отримує об’єкт партнера на виході.
- Узагальнений клас java.util.concurrent.Exchanger<V> реалізує синхронізатор типу обмінник.
  - Для ініціалізації використовується конструктор Exchanger().
  - Для обміну далі береться один з методів:

- `V exchange(V x)`: Waits for another thread to arrive at this exchange point (unless the calling thread is interrupted), and then transfers the given object to it, receiving the other thread’s object in return. If another thread is already waiting at the exchange point, it’s resumed for thread scheduling purposes and receives the object passed in by the calling thread. The current thread returns immediately, receiving the object passed to the exchanger by the other thread. This throws `InterruptedException` when the calling thread is interrupted.
- `V exchange(V x, long timeout, TimeUnit unit)`: This method is equivalent to the previous method except that it lets you specify how long the calling thread is willing to wait. It throws `TimeoutException` when this timeout expires while the thread is waiting.

```
import java.util.ArrayList;
import java.util.List;

import java.util.concurrent.Exchanger;

public class ExchangerDemo
{
   static Exchanger<DataBuffer> exchanger = new Exchanger<DataBuffer>();

   static DataBuffer initialEmptyBuffer = new DataBuffer();
   static DataBuffer initialFullBuffer = new DataBuffer("I");

   public static void main(String[] args)
   {
      class FillingLoop implements Runnable
      {
         int count = 0;

         @Override
         public void run()
         {
            DataBuffer currentBuffer = initialEmptyBuffer;
            try
            {
               while (true)
               {
                  addToBuffer(currentBuffer);
                  if (currentBuffer.isFull())
                  {
                     System.out.println("filling thread wants to exchange");
                     currentBuffer = exchanger.exchange(currentBuffer);
                     System.out.println("filling thread receives an exchange");
                  }
               }
            }
            catch (InterruptedException ie)
            {
               System.out.println("filling thread interrupted");
            }
         }

         void addToBuffer(DataBuffer buffer)
         {
            String item = "NI" + count++;
            System.out.println("Adding: " + item);
            buffer.add(item);
         }
      }
```

Використання обмінників для очистки буфера

- Головний потік створює обмінник та пару буферів за допомогою статичних ініціалізаторів екземпляру.
- Потім виконується інстанціювання класу EmptyingLoop та локального класу FillingLoop; ці runnable-об’єкти передаються екземплярам new Thread, чиї потоки потім запускаються.

```
class EmptyingLoop implements Runnable
{
   @Override
   public void run()
   {
      DataBuffer currentBuffer = initialFullBuffer;
      try
      {
         while (true)
         {
            takeFromBuffer(currentBuffer);
            if (currentBuffer.isEmpty())
            {
               System.out.println("emptying thread wants to exchange");
               currentBuffer = exchanger.exchange(currentBuffer);
               System.out.println("emptying thread receives an exchange");
            }
         }
      }
      catch (InterruptedException ie)
      {
         System.out.println("emptying thread interrupted");
      }
   }

   void takeFromBuffer(DataBuffer buffer)
   {
      System.out.println("taking: " + buffer.remove());
   }
}
new Thread(new EmptyingLoop()).start();
new Thread(new FillingLoop()).start();
   }
}
```

- Метод run() кожного runnable заходить у нескінченний цикл, який repeatedly додає в або видаляє з its buffer.
- Коли буфер повний або порожній, обмінник використовується для обміну буферів, а заповнення чи очищення продовжується.

```
class DataBuffer
{
   private final static int MAXITEMS = 10;

   private List<String> items = new ArrayList<String>();

   DataBuffer()
   {
   }

   DataBuffer(String prefix)
   {
      for (int i = 0; i < MAXITEMS; i++)
      {
         String item = prefix+i;
         System.out.printf("Adding %s%n", item);
         items.add(item);
      }
   }

   void add(String s)
   {
      if (!isFull())
         items.add(s);
   }

   boolean isEmpty()
   {
      return items.size() == 0;
   }

   boolean isFull()
   {
      return items.size() == MAXITEMS;
   }
   String remove()
   {
      if (!isEmpty())
         return items.remove(0);
      return null;
   }
}
```

```
Adding I0
Adding I1
Adding I2
Adding I3
Adding I4
Adding I5
Adding I6
Adding I7
Adding I8
Adding I9
taking: I0
taking: I1
taking: I2
taking: I3
taking: I4
taking: I5
taking: I6
taking: I7
taking: I8
taking: I9
emptying thread wants to exchange
```

```
Adding: NI0
Adding: NI1
Adding: NI2
Adding: NI3
Adding: NI4
Adding: NI5
Adding: NI6
Adding: NI7
Adding: NI8
Adding: NI9
filling thread wants to exchange
filling thread receives an exchange
emptying thread receives an exchange
Adding: NI10
taking: NI0
Adding: NI11
taking: NI1
Adding: NI12
```

Семафори

- Семафор підтримує набір дозволів (*permits*) для обмеження кількості потоків, які матимуть доступ до обмежених ресурсів.
  - Потік намагається отримати дозвіл, якщо для доступних блоків не має його, поки інший потік не releases a permit.
  - Семафори, поточні значення яких можуть інкрементуватись past 1, відомі як ***counting semaphores***.
  - семафори, чиї поточні значення можуть бути лише 0 або 1, називаються ***бінарнимим семафорами***, або ***м'ютексами (mutex)***. У жодному разі значення не можуть бути від'ємними.
- Клас java.util.concurrent.Semaphore реалізує цей синхронізатор та описує семафор як об'єкт, що підтримує набір дозволів.
  - Ініціалізація семафору відбувається за допомогою виклику конструктора **Semaphore(int permits)**, у якому permits задає кількість доступних дозволів.
  - У результаті *fairness policy* семафору задається як false (unfair).
  - Як альтернативу, можна викликати конструктор **Semaphore(int permits, boolean fair)**, щоб задати semaphore's fairness значення true (fair).
  - Коли fairness задається як false, клас Semaphore не гарантує порядку отримання дозволів потоками.

Семафори та fairness

- Коли fairness задається як false, клас Semaphore не гарантує порядку отримання дозволів потоками.
  - Зокрема, дозволяється barging; потоку, що викликає acquire(), може виділятись дозвіл

> In particular, *barging* is permitted; that is, a thread invoking `acquire()` can be allocated a permit ahead of a thread that has been waiting; logically the new thread places itself at the head of the queue of waiting threads. When `fair` is set to `true`, the semaphore guarantees that threads invoking any of the `acquire()` methods are selected to obtain permits in the order in which their invocation of those methods was processed (first-in, first-out; or FIFO). Because FIFO ordering necessarily applies to specific internal points of execution within these methods, it's possible for one thread to invoke `acquire()` before another thread, but reach the ordering point after the other thread and similarly upon return from the method. Also, the untimed `tryAcquire()` methods don't honor the fairness setting; they'll take any available permits.
>
> Generally, semaphores used to control resource access should be initialized as fair to ensure that no thread is starved out from accessing a resource. When using semaphores for other kinds of synchronization control, the throughput advantages of unfair ordering often outweigh fairness considerations.

Методи класу Semaphore

- `void acquire()`: Acquires a permit from this semaphore, blocking until one is available or the calling thread is interrupted. `InterruptedException` is thrown when interrupted.
- `void acquire(int permits)`: Acquires `permits` permits from this semaphore, blocking until they are available or the calling thread is interrupted. `InterruptedException` is thrown when interrupted; `IllegalArgumentException` is thrown when `permits` is less than zero.
- `void acquireUninterruptibly()`: Acquires a permit, blocking until one is available.
- `void acquireUninterruptibly(int permits)`: Acquires `permits` permits, blocking until they are all available. `IllegalArgumentException` is thrown when `permits` is less than zero.
- `int availablePermits()`: Returns the current number of available permits. This method is useful for debugging and testing.
- `int drainPermits()`: Acquires and returns a count of all permits that are immediately available.
- `int getQueueLength()`: Returns an estimate of the number of threads waiting to acquire permits. The returned value is only an estimate because the number of threads may change dynamically while this method traverses internal data structures. This method is designed for use in monitoring system state and not for synchronization control.
- `boolean hasQueuedThreads()`: Queries whether any threads are waiting to acquire permits. Because cancellations may occur at any time, a true return value doesn't guarantee that another thread will ever acquire permits. This method is designed primarily for use in monitoring system state. It returns true when there may be other waiting threads.
- `boolean isFair()`: Returns the fairness setting (true for fair and false for unfair).
- `void release()`: Releases a permit, returning it to the semaphore. The number of available permits is increased by one. If any threads are trying to acquire a permit, one thread is selected and given the permit that was just released. That thread is reenabled for thread scheduling purposes.
- `void release(int permits)`: Releases `permits` permits, returning them to the semaphore. The number of available permits is increased by `permits`. If any threads are trying to acquire permits, one is selected and given the permits that were just released. If the number of available permits satisfies that thread's request, the thread is reenabled for thread scheduling purposes; otherwise, the thread will wait until sufficient permits are available. If there are permits available after this thread's request has been satisfied, those permits are assigned to other threads trying to acquire permits. `IllegalArgumentException` is thrown when `permits` is less than zero.
- `String toString()`: Returns a string identifying this semaphore as well as its state. The state, in brackets, includes the string literal `"Permits ="` followed by the number of permits.
- `boolean tryAcquire()`: Acquires a permit from this semaphore but only when one is available at the time of invocation. Returns true when the permit was acquired. Otherwise, returns immediately with value false.

- `boolean tryAcquire(int permits)`: Acquires `permits` permits from this semaphore, but only when they are available at the time of invocation. Returns true when the permits were acquired. Otherwise, returns immediately with value false. `IllegalArgumentException` is thrown when `permits` is less than zero.
- `boolean tryAcquire(int permits, long timeout, TimeUnit unit)`: Like the previous method, but the calling thread waits when `permits` permits aren't available. The wait ends when the permits become available, the timeout expires, or the calling thread is interrupted, in which case `InterruptedException` is thrown.
- `boolean tryAcquire(long timeOut, TimeUnit unit)`: Like `tryAcquire(int permits)`, but the calling thread waits until a permit is available. The wait ends when the permit becomes available, the timeout expires, or the calling thread is interrupted, in which case `InterruptedException` is thrown.

Використання counting semaphore для керування доступом до пулу item-ів

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Semaphore;
public class SemaphoreDemo
{
   public static void main(String[] args)
   {
      final Pool pool = new Pool();
      Runnable r = new Runnable()
                   {
                      @Override
                      public void run()
                      {
                         String name = Thread.currentThread().getName();
                         try
                         {
                            while (true)
                            {
                               String item;
                               System.out.println(name + " acquiring " +
                                                  (item = pool.getItem()));
                               Thread.sleep(200 + (int) (Math.random() * 100));
                               System.out.println(name + " putting back " + item);
                               pool.putItem(item);
                            }
                         }
                         catch (InterruptedException ie)
                         {
                            System.out.println(name + "interrupted");
                         }
                      }
                   };
      ExecutorService[] executors = new ExecutorService[Pool.MAX_AVAILABLE+1];
      for (int i = 0; i < executors.length; i++)
      {
         executors[i] = Executors.newSingleThreadExecutor();
         executors[i].execute(r);
      }
   }
}
```

- Головний потік створює ресурсний пул, runnable for repeatedly acquiring and putting back resources, та масив екзекуторів.
  - Кожному екзекутору сказано виконувати runnable-задачу.
- Методи String getItem() і void putItem(String item) класу Pool отримують та повертають рядкові ресурси.
- До отримання елементу в getItem() викликаючий потік повинен одержати дозвіл від семафору, який гарантує, що елемент (item) доступний для використання.
- Коли потік завершує роботу з елементом, він викликає putItem(String), який повертає елемент в пул, а потім releases a permit to the semaphore, which lets another thread acquire that item.

```
final class Pool
{
   public static final int MAX_AVAILABLE = 10;

   private Semaphore available = new Semaphore (MAX_AVAILABLE, true);

   private String[] items;

   private boolean[] used = new boolean[MAX_AVAILABLE];

   Pool()
   {
      items = new String[MAX_AVAILABLE];
      for (int i = 0; i < items.length; i++)
         items[i] = "I" + i;
   }

   String getItem() throws InterruptedException
   {
      available.acquire();
      return getNextAvailableItem();
   }

   void putItem(String item)
   {
      if (markAsUnused(item))
         available.release();
   }

   private synchronized String getNextAvailableItem()
   {
      for (int i = 0; i < MAX_AVAILABLE; ++i)
      {
         if (!used[i])
         {
            used[i] = true;
            return items[i];
         }
      }
      return null; // not reached
   }
```

- Коли викликається acquire(), синхронізаційного замка (lock) немає, оскільки це заважатиме елементу повернутись у пул.
  - Проте методи String getNextAvailableItem() та boolean markAsUnused(String item) синхронізовані, щоб підтримувати цілісність пулу.
  - Семафор інкапсулює синхронізацію для заборони доступу до пулу без синхронізації, необхідної для підтримки цілісності пулу.

```
   private synchronized boolean markAsUnused(String item)
   {
      for (int i = 0; i < MAX_AVAILABLE; ++i)
      {
         if (item == items[i])
         {
            if (used[i])
            {
               used[i] = false;
               return true;
            }
            else
               return false;
         }
      }
      return false;
   }
}
```

Результат виводу

```
pool-2-thread-1 acquiring I0
pool-4-thread-1 acquiring I1
pool-6-thread-1 acquiring I2
pool-8-thread-1 acquiring I3
pool-10-thread-1 acquiring I4
pool-1-thread-1 acquiring I5
pool-3-thread-1 acquiring I6
pool-5-thread-1 acquiring I7
pool-7-thread-1 acquiring I8
pool-9-thread-1 acquiring I9
pool-6-thread-1 putting back I2
pool-11-thread-1 acquiring I2
pool-2-thread-1 putting back I0
pool-6-thread-1 acquiring I0
pool-4-thread-1 putting back I1
pool-2-thread-1 acquiring I1
pool-1-thread-1 putting back I5
pool-4-thread-1 acquiring I5
pool-5-thread-1 putting back I7
pool-1-thread-1 acquiring I7
pool-8-thread-1 putting back I3
pool-5-thread-1 acquiring I3
pool-7-thread-1 putting back I8
pool-8-thread-1 acquiring I8
pool-10-thread-1 putting back I4
pool-7-thread-1 acquiring I4
pool-9-thread-1 putting back I9
pool-10-thread-1 acquiring I9
```

Синхронізатори в цілому