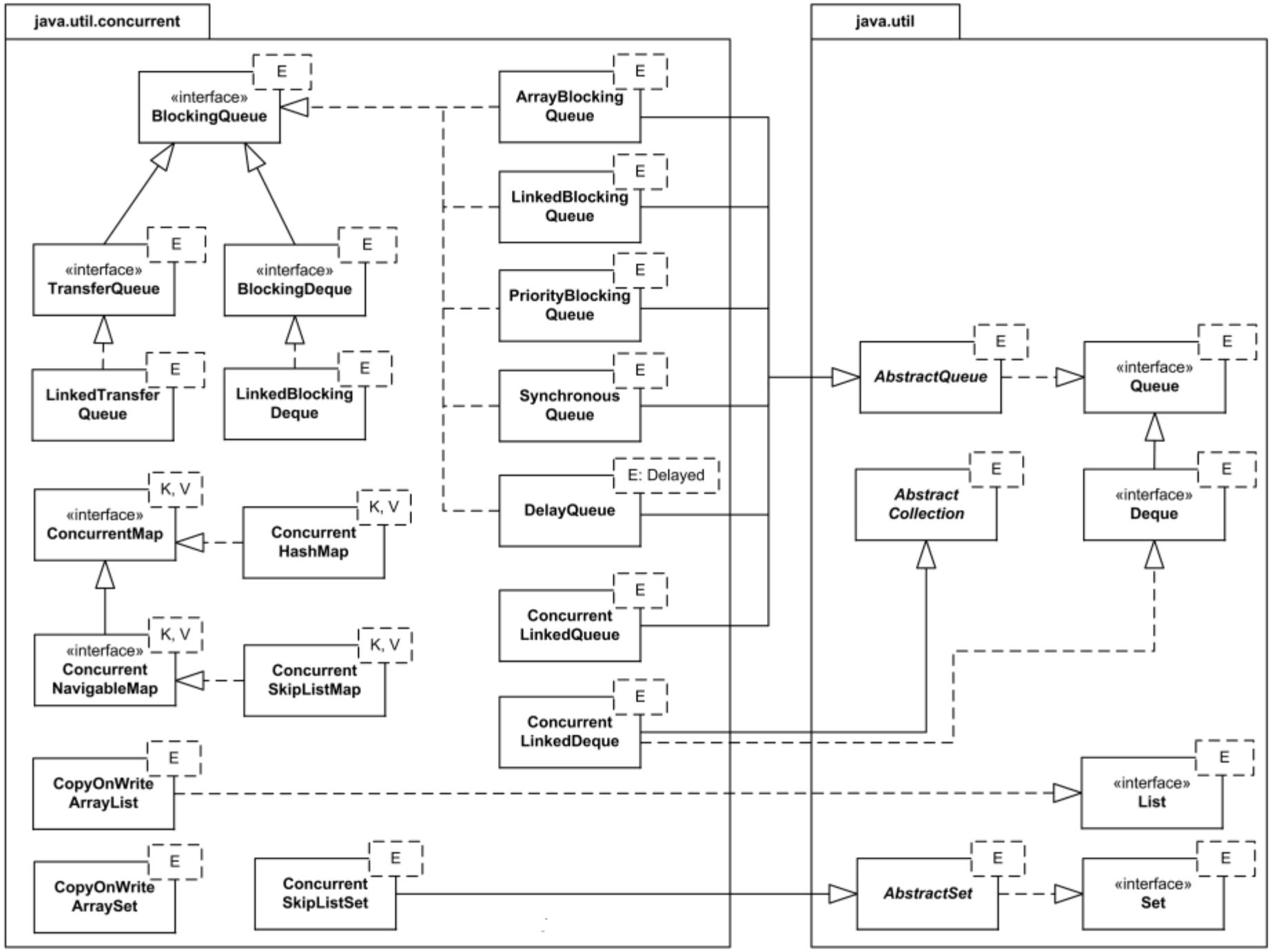
Лекція 2. Блокуючі черги та деки.

Тема 1. Основні задачі та характеристики інтерфейсів BlockingQueue, TransferQueue, BlockingDeque.

Тема 2. Методи обробки запитів на отримання, додавання та перегляд задач у блокуючій черзі або деці.

Тема 3. Методи класів ArrayBlockingQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedTransferQueue, PriorityBlockingQueue, SynchronousQueue.

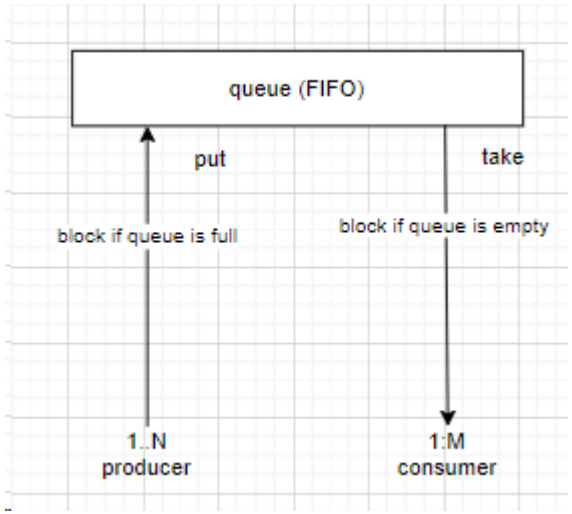


Classes

1. AbstractExecutorService
2. ArrayBlockingQueue
3. ConcurrentHashMap
4. ConcurrentLinkedDeque
5. ConcurrentLinkedQueue
6. ConcurrentSkipListMap
7. ConcurrentSkipListSet
8. CopyOnWriteArrayList
9. CopyOnWriteArraySet
10. CountDownLatch
11. CyclicBarrier
12. DelayQueue
13. Exchanger
14. ExecutorCompletionService
15. Executors
16. ForkJoinPool
17. ForkJoinTask
18. ForkJoinWorkerThread
19. FutureTask
20. LinkedBlockingDeque
21. LinkedBlockingQueue
22. LinkedTransferQueue
23. Phaser
24. PriorityBlockingQueue
25. RecursiveAction
26. RecursiveTask
27. ScheduledThreadPoolExecutor
28. Semaphore
29. SynchronousQueue
30. ThreadLocalRandom
31. ThreadPoolExecutor
32. ThreadPoolExecutor.AbortPolicy
33. ThreadPoolExecutor.CallerRunsPolicy
34. ThreadPoolExecutor.DiscardOldestPolic
    y
35. ThreadPoolExecutor.DiscardPolicy

Array Blocking Queue

Логіка «постачальник» - «споживача»: «постачальник» даних (producer) та «споживач» даних (consumer) мають наступні функції: «постачальник» - додає дані до деякого сховища, «споживач» - забирає дані зі сховища. Якщо в сховище немає даних - «споживач» чекає; якщо в сховище немає місця - «постачальник» чекає.



Розглянемо реалізацію ArrayBlockingQueue з методами класу Монітор: synchronized, wait(), notifyAll()

```
import java.util.LinkedList;

public class ArrayBlockingQueue<T> {
   private final LinkedList<T> queue;
   private final int capacity;

   public ArrayBlockingQueue(int capacity) {
      this.capacity = capacity;
      this.queue = new LinkedList<>();
   }

   public synchronized void put(T item) throws
InterruptedException {
      while (queue.size() == capacity) {
            wait();
      }
      queue.add(item);
      notifyAll();
   }

   public synchronized T take() throws InterruptedException {
      while (queue.isEmpty()) {
            wait();
      }
      T item = queue.remove();
        notifyAll();
      return item;
   }

   public synchronized int size() {
      return queue.size();
   }
}
```

```
  public static void main(String[] args) {
      final int capacity = 5;
      ArrayBlockingQueue<Integer> queue = new ArrayBlockingQueue<>(capacity);
Thread producerThread = new Thread(() -> {
         try {
            for (int i = 0; i < 10; i++) {
               queue.put(i);
               System.out.println("Produced: " + i);
               Thread.sleep(1000);
            }
         } catch (InterruptedException e) {
            e.printStackTrace();
         }

      });
      Thread consumerThread = new Thread(() -> {
         try {
            for (int i = 0; i < 10; i++) {
               int item = queue.take();
               System.out.println("Consumed: " + item);
               Thread.sleep(2000);
            }
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      });
      producerThread.start();
      consumerThread.start();

try {
         producerThread.join();
         consumerThread.join();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
```

1) Скільки повідомлень постачальник відправить в чергу?

Проста реалізацію блокуючої черги з використанням LinkedList. Блокуюча черга це тип структури даних, який підтримує операції вставлення та видалення елементів, також управляє розміром черги, не дозволяючи їй переповнюватися.

Поля класу:

private final LinkedList<T> queue; - об'єкт черги, яка використовується для зберігання елементів.

private final int capacity; - максимальна кількість елементів, яку може вмістити черга.

Конструктор:

public ArrayBlockingQueue(int capacity) - конструктор приймає один аргумент capacity, який встановлює максимальну ємність черги, і ініціалізує чергу як новий LinkedList.

Методи:

public synchronized void put(T item) - Метод для додавання елемента в чергу. Якщо черга заповнена (тобто розмір черги дорівнює ємності), потічний потік буде чекати, поки в черзі не з'явиться місце. Після додавання елемента метод notifyAll() викликається для пробудження інших потоків, які можуть очікувати на можливість додати або взяти елемент.

public synchronized T take() - Метод для взяття та видалення першого елемента з черги. Якщо черга порожня, поточний потік чекатиме, поки не з'явиться, щонайменше, один елемент. Після видалення елемента метод notifyAll() також викликається для пробудження чекаючих потоків.

public synchronized int size() - Метод повертає поточний розмір черги.

Ключовим аспектом цього класу є використання ключового слова synchronized для методів put, take та size, що забезпечує взаємне блокування потоків для уникнення одночасного доступу до черги різними потоками, що може призвести до логічних помилок або невірного стану даних. Методи wait та notifyAll використовуються для керування потоками, які очікують на можливість додати або взяти елементи з черги, коли вона повна або порожня.

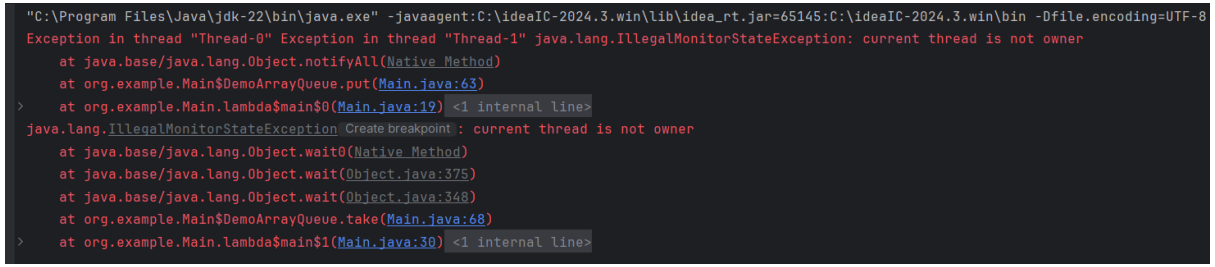
Synchronized

Метод wait() використовується в put() і take(), щоб призупинити виконання потоку, коли певні умови не виконуються (наприклад, черга заповнена або порожня).

Для виклику wait() потік, що викликає, повинен утримувати внутрішнє блокування або блокування монітора об’єкта (тобто метод має бути визначений ключовим словом **Synchronized**).

Подібним чином, notifyAll() викликається, щоб пробудити всі інші потоки, які очікують на моніторі об’єкта. Цей метод також вимагає, щоб викликаючий потік утримував внутрішнє блокування об’єкта, для якого він викликається.

Якщо залишити wait() без **Synchronized - Без включення цих викликів у синхронізований блок або без синхронізації самих методів потік, який викликає wait() або notifyAll(), не утримує необхідного блокування на моніторі об’єкта. Це призводить до IllegalMonitorStateException.**



"монітор об'єкта"

в Java "монітор об'єкта" — це механізм, який забезпечує взаємне блокування (synchronization), що дозволяє керувати доступом до спільних ресурсів різними потоками. Кожен об'єкт в Java має монітор, який використовується для реалізації блоків синхронізації.

Монітори виконують дві основні функції:

Lock (Блокування):

Коли потік входить у синхронізований блок або метод, він автоматично отримує "замок" на моніторі цього об'єкта.

Якщо інший потік намагається виконати синхронізований блок або метод того ж об'єкта, він буде заблокований до тих пір, поки перший потік не вийде з блоку або методу і не звільнить монітор.

Wait Set (Набір очікування):

Коли потік викликає метод wait() всередині синхронізованого контексту, він відпускає замок монітора та переміщується до набору очікування цього монітора.

Потоки у наборі очікування залишаються заблокованими до тих пір, поки інший потік не викликає notify() або notifyAll() на тому ж об'єкті, в результаті чого один або всі потоки у наборі будуть пробуджені і спробують знову отримати замок на моніторі.

Отже, монітор об'єкта допомагає забезпечити, що потоки не входять у критичні секції коду, де вони могли б взаємодіяти з спільними даними у спосіб, який міг би призвести до непослідовності або помилок програми. Це дає можливість створювати додатки, які правильно і ефективно використовують багатопоточність, мінімізуючи ризики конфліктів даних та забезпечуючи коректну координацію між потоками.

Спільний об'єкт

```
public static void commonobjectdemo()
   {
      int numberOfThreads = 10;
      Thread[] threads = new Thread[numberOfThreads];

      for (int t = 0; t < numberOfThreads; t++) {
         threads[t] = new Thread(() -> {
            for (int i = 0; i < 10000; i++) {
                  counter++;
}
         });

         threads[t].start();
      }



      for (Thread thread : threads) {
         try {
            thread.join();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }

      System.out.println("Final counter value: " + counter);
   }
```

Яке значення counter слід очікувати?

Якщо не використовувати synchronized у багатопоточному програмуванні для ресурсів, які доступні для декількох потоків, можуть виникнути різні типи конфліктів або помилок. Один з таких конфліктів називається змаганням за ресурси (race condition).

Приклад конфлікт:

Уявімо, що ми маємо блокуючу чергу, як у попередньому прикладі коду, але методи put() і take() не синхронізовані.

Маймо два потоки:

Потік Producer, який виконує метод put(item) для додавання елемента до черги.

Потік Consumer, який виконує метод take() для забирання елемента з черги.

Сценарій змагання:

Потік А перевіряє розмір черги (queue.size()) і бачить, що черга не повна.

В цей самий час, потік В також перевіряє, чи черга не порожня, і бачить, що вона непорожня.

Потік А додає елемент в кінець черги.

Майже одночасно, потік В вирішує взяти і видалити елемент з початку черги.

Якщо ці операції виконуються без належної синхронізації, можлива ситуація, коли потік В спробує забрати елемент із черги до того, як потік А встигне додати новий елемент, навіть якщо поміж перевіркою стану і додаванням/забиранням елементу в черзі минула мілісекунда. Це може призвести до спроби звернення до неіснуючого елемента та викликати помилку часу виконання.

public interface BlockingQueue<E> extends Queue<E>.

Implementing Classes: ArrayBlockingQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedTransferQueue, PriorityBlockingQueue, SynchronousQueue

Черга, яка підтримує операції, які очікують, поки черга стане непорожньою під час отримання елемента, і чекають, поки в черзі звільниться місце під час збереження елемента. Підтримує чотири форми обробки операцій отримання/додавання/перегляду даних:

- ✓ перший тип – методи генерують виняткову ситуацію, якщо дію не виконати.
- ✓ другий - повертають спеціальне значення (або null, або false, залежно від операція).
- ✓ третій - блокує поточний потік на невизначений термін, доки операція не завершиться успішно.
- ✓ Четвертий - блокує лише протягом певного максимального часу, перш ніж відмовитися.

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	*not applicable*	*not applicable*

public class DelayQueue<E extends Delayed> extends AbstractQueue<E> implements BlockingQueue<E>

DelayQueue — це спеціалізована реалізація черги із затримкою в Java, яка заснована на інтерфейсі BlockingQueue. Елементи, які додаються до DelayQueue, мають певний час затримки, і ці елементи стають доступними лише після завершення їхньої затримки. Елементи у черзі мають реалізовувати інтерфейс Delayed.

Термін дії елемента закінчився, якщо його метод ***getDelay(TimeUnit. NANOSECONDS)*** повертає значення, менше або дорівнює нулю.

Методи, які працюють узгоджено відповідно часу затримки

poll()

poll(long, TimeUnit)

take()

remove()

Усі інші методи діють як на прострочені, так і на непрострочені елементи. Наприклад, метод size() повертає кількість усіх елементів.

Метод peek() може повертати (не нульовий) заголовок, навіть якщо take() блокував би очікування закінчення терміну дії цього елемента.

Ця черга не дозволяє нульові елементи.

```
static class DelayedItem implements Delayed {
    private final String item;
    private final long expirationTime;

    public DelayedItem(String item, long delay, TimeUnit unit) {
        this.item = item;
        this.expirationTime = System.currentTimeMillis() +
TimeUnit.MILLISECONDS.convert(delay, unit);
    }

    @Override
    public long getDelay(TimeUnit unit) {
        long delay = expirationTime - System.currentTimeMillis();
        return unit.convert(delay, TimeUnit.MILLISECONDS);
    }

    @Override
    public int compareTo(Delayed o) {
        if (this.expirationTime < ((DelayedItem)o).expirationTime) {
            return -1;
        }
        if (this.expirationTime > ((DelayedItem)o).expirationTime) {
            return 1;
        }
        return 0;
    }

    public String getItem() {
        return item;
    }
}
```

Клас DelayedItem реалізує інтерфейс Delayed і представляє елемент з затримкою.
Конструктор DelayedItem приймає рядок (елемент), час затримки і одиницю часу.
Час затримки перетворюється у мілісекунди і додається до поточного часу для визначення моменту, коли елемент стане доступним.
Метод getDelay повертає час, що залишився до активації елемента.

```
public static void testdelayqueue()
   {

       DelayQueue<DelayedItem> queue = new DelayQueue<>();

      queue.put(new DelayedItem("Item1", 5, TimeUnit.SECONDS));
       queue.put(new DelayedItem("Item2", 10, TimeUnit.SECONDS));
       queue.put(new DelayedItem("Item3", 1, TimeUnit.SECONDS));


       while (!queue.isEmpty()) {
           DelayedItem item = null;
           try {
               item = queue.take();
           } catch (InterruptedException e) {
               throw new RuntimeException(e);
           }
           System.out.println("Оброблено: " + item.getItem() + " на час: " +
System.currentTimeMillis());
       }
   }
```

У якому порядку отримаємо елементи?

Метод take() в DelayQueue — це блокуючий метод, який використовується для вилучення елементів із черги. Цей метод чекає, поки не стане доступним елемент із черги, тобто елемент, час затримки якого сплив. Коли елемент може бути вилучений (тобто його затримка закінчилася), take() повертає цей елемент і видаляє його з черги.

Як працює метод take() у DelayQueue:
Чекання доступності: Коли метод take() викликається, він перш за все перевіряє, чи є в черзі елементи, доступні для вилучення, тобто ті елементи, чия затримка вже спливла.
Робота з блокуваннями: Якщо жоден елемент не готовий (тобто всі елементи все ще мають невід'ємний час затримки), потік, який викликає take(), блокується і переходить у режим очікування. Це очікування триватиме доти, доки затримка першого елемента в черзі не спливе.
Пробудження та вилучення: Коли настає час затримки елемента, DelayQueue автоматично "пробуджує" потік, який виконує операцію take(). Метод take() тоді повертає цей елемент, вилучаючи його з черги.

```
public String take() throws InterruptedException {
    while (true) {
        DelayedItem item = queue.peek();
        if (item != null) {
            long delay = item.getDelay();
            if (delay <= 0) {
                return queue.poll().getItem();
            }
            synchronized (this) {
                TimeUnit.MILLISECONDS.sleep(delay);  // Чекаємо, поки
елемент не стане досяжним
            }
        }
    }
}
```

Клас PriorityQueue<E> у Java - це структура даних, яка дозволяє зберігати елементи та вилучати їх згідно з природнім порядком або за допомогою порівнювача (Comparator), якщо такий вказаний. Основна особливість цього класу полягає в тому, що він реалізує чергу з пріоритетами, де із черги завжди вилучається елемент із найвищим пріоритетом. Для PriorityQueue мінімальний елемент має найвищий пріоритет, якщо не вказано інше через компаратор.

Ось основні характеристики PriorityQueue:

PriorityQueue наслідується від AbstractQueue і реалізує інтерфейс Queue.

Дані в PriorityQueue зазвичай зберігаються у вигляді бінарної купи. Ця структура даних забезпечує ефективне вилучення мінімального елементу.

Конструктори

PriorityQueue пропонує декілька конструкторів:

PriorityQueue(): створює PriorityQueue із початковою місткістю 11.

PriorityQueue(int initialCapacity): створює PriorityQueue із заданою початковою місткістю.

PriorityQueue(int initialCapacity, Comparator<? super E> comparator): створює PriorityQueue із заданою початковою місткістю та компаратором.

PriorityQueue(Collection<? extends E> c): створює PriorityQueue, що містить елементи зі специфікованої колекції.

PriorityQueue(PriorityQueue<? extends E> c), PriorityQueue(SortedSet<? extends E> c): конструктори для створення PriorityQueue на базі інших колекцій.

Методи

Основні оперативні методи включають offer(E e), poll(), peek() і remove(Object o).

offer(E e): вставляє елемент у чергу.

poll(): вилучає та повертає головний елемент черги, якщо черга не порожня.

peek(): повертає, але не вилучає головний елемент черги.

Методи iterator() та sort() дозволяють ітерувати через елементи та сортувати купу відповідно.

Важливі Відомості

Ітерація на PriorityQueue не гарантує витягування елементів у порядку їх пріоритету. Якщо необхідно перебрати купу за пріоритетом, слід вилучати елементи за допомогою poll().

PriorityQueue не допускає вставки null.

Ця структура не є синхронізованою. Для багатопоточного використання рекомендується застосування методів з Collections.synchronizedCollection() або використання конкурентних структур, таких як PriorityBlockingQueue.

PriorityQueue часто використовується там, де елементи повинні оброблятися у залежності від їх пріоритету, наприклад, в алгоритмах як Dijkstra, урегулювання завдань за пріоритетом, та ін.

Який буде очікуваний результат?

```
import java.util.Comparator;
import java.util.PriorityQueue;

public class SimplePriorityBlockingQueue<T> {
    private final PriorityQueue<T> queue;
    private final int capacity;

    public SimplePriorityBlockingQueue(int capacity, Comparator<?
super T> comparator) {
        this.capacity = capacity;
        this.queue = new PriorityQueue<>(comparator);
    }

    public synchronized void put(T item) throws
InterruptedException {
        while (queue.size() == capacity) {
            wait();
        }
        queue.add(item);
        notifyAll();
}

    public synchronized T take() throws InterruptedException {
        while (queue.isEmpty()) {
            wait();
        }
        T item = queue.poll();
        notifyAll();
       return item;
    }
}
```

```
public static void DemoPriorityQueue()
 {
    SimplePriorityBlockingQueue<Integer> queue = new SimplePriorityBlockingQueue<>(5, Integer::compare);
    Thread producer = new Thread(() -> {
       int[] numbers = {5, 1, 3, 2, 4};
       for (int number : numbers) {
          try {
             System.out.println("Producer putting value: " + number);
             queue.put(number);
             Thread.sleep(100);
          } catch (InterruptedException e) {
             Thread.currentThread().interrupt();
          }
       }
    });
    Thread consumer = new Thread(() -> {
       try {
          Thread.sleep(500); //
       } catch (InterruptedException e) {
          Thread.currentThread().interrupt();
       }

       while (!Thread.interrupted()) {
          try {
             Integer value = queue.take();
             System.out.println("Consumer took value: " + value);
             Thread.sleep(150);
          } catch (InterruptedException e) {
             Thread.currentThread().interrupt();
             break;
          }
       }
    });

    producer.start();
    consumer.start();
    try {
       producer.join();
       consumer.join();
    } catch (InterruptedException e) {
       e.printStackTrace();
    }
 }
}
```

У Java, використання String::compare безпосередньо як посилання на метод для порівняння об'єктів String неможливе, оскільки клас String не має статичного методу під назвою compare. Проте, клас String у Java реалізує інтерфейс Comparable, що означає, що кожен екземпляр String має метод compareTo, який порівнює String з іншим String за лексикографічним порядком.

Коли вам потрібен Comparator<String> для структури, як PriorityQueue, або будь-якої іншої колекції, що вимагає явних компараторів, ви можете використати метод compareTo у виразі лямбда або можете використати посилання на метод Comparator.naturalOrder() для природного упорядкування рядків. Ось як ви можете їх використовувати:

```
PriorityQueue<String> queue = new PriorityQueue<>((s1, s2) -> s1.compareTo(s2));
```

Використання посилання на метод Comparator.naturalOrder(), який повертає компаратор, що враховує природний порядок (який для рядків є лексикографічний порядок):

```
PriorityQueue<String> queue = new PriorityQueue<>(Comparator.naturalOrder());
```

Обидва підходи впорядковують об'єкти String у черзі з пріоритетами лексикографічно. Якщо вам потрібно забезпечити порядок, незалежний від регістру, ви можете використати String::compareToIgnoreCase або визначити специфічний компаратор:

SynchronousQueue

SynchronousQueue - це особливий вид черги, який використовується в Java з пакетом java.util.concurrent. На відміну від інших черг, SynchronousQueue не має жодної внутрішньої ємності, навіть вмістимості в один елемент. Це означає, що вставка елемента у чергу буде блокована до тих пір, поки інший потік не вилучить цей елемент; аналогічно, вилучення з черги буде блоковане, поки інший потік не вставить елемент.

Призначення SynchronousQueue:
Ця черга типово використовується в системах з високим ступенем спільної взаємодії між потоками, де потік, що створює дані, має безпосередньо передавати їх потоку, який їх обробляє. Завдяки цьому забезпечується дуже тісна синхронізація між потоками, і потік-споживач може одразу обробити дані, як тільки ті стануть до нього доступними.

Особливості SynchronousQueue:
Блокування вставки та вилучення. Кожна операція вставки блокується до тих пір, поки інший потік не вилучить елемент, і навпаки, кожна операція вилучення блокується до тих пір, поки інший потік не вставить елемент.
Пряма передача. SynchronousQueue оптимізована для прямої передачі даних між потоками, що ефективно використовує системні ресурси.
Використання у concurrency-конструкціях. Черга часто використовується в реалізації виконавчих служб (Executor Services), зокрема у пулах потоків, таких як CachedThreadPool.

```
public static class SynchronousQueueObjectLock<T> {
    private T item = null;
    private boolean hasItem = false;
    private final Object lock = new Object(); // Lock for
synchronization

    public void put(T value) throws InterruptedException {
        synchronized (lock) {
            while (hasItem) {
                System.
out.println(Thread.currentThread().getName()+ "Producer is
waiting");
                lock.wait();
            }
            item = value; // Store item
            hasItem = true; // Mark item as available
            lock.notifyAll(); // Notify waiting consumers
        }
    }

    public T take() throws InterruptedException {
        synchronized (lock) {
            while (!hasItem) { // Wait until an item is available
                System. out.println("Consumer is waiting");
                lock.wait();
            }
            T value = item; // Retrieve item
            item = null; // Clear item
            hasItem = false; // Mark as empty
            lock.notifyAll(); // Notify waiting producers
            return value;
        }
    }
}
```

```
public static class MySynchronousQueue<T> {

        private T item = null;

        public synchronized void put(T value) throws
InterruptedException {
            while (item != null) {

                System.
out.println(Thread.currentThread().getName()+ "Producer is
waiting");
                wait();
            }
            item = value;
            notifyAll();
        }

        public synchronized T take() throws
InterruptedException {
            while (item == null) {
                System. out.println("Consumer is waiting");
                wait();
            }
            T value = item;
            item = null;
            notifyAll();
            return value;
        }
    }
```

1. ***Гарантія, що тільки один потік змінює спільну змінну***
Методи put() і take() використовують synchronized (lock) {} для того, щоб лише один потік за раз мав доступ до змінної item.
Це запобігає стану гонки (race condition), коли кілька потоків намагаються одночасно змінити item.

2. ***Блокування та очікування (lock.wait())***
Якщо викликається put(), коли item вже містить значення (hasItem == true), потік-виробник чекає.
Аналогічно, якщо викликається take(), коли item ще не доданий (hasItem == false), потік-споживач чекає.
Виклик lock.wait(); змушує потік звільнити блокування і чекати, поки інший потік викличе lock.notifyAll();.
3. Пробудження потоку, що очікує (lock.notifyAll())
Після того як put() додає елемент у item, він викликає lock.notifyAll();, щоб розбудити будь-який потік, який чекає на take().
Аналогічно, після того як take() отримує та очищає item, він викликає lock.notifyAll();, щоб розбудити будь-який потік, який чекає на put().

Об'єкт lock використовується як спеціальний монітор для синхронізації.
synchronized (this), блокує весь екземпляр класу MySynchronousQueue, що в більшості випадків допустимо. Однак використання окремого об'єкта lock надає більшу гнучкість у разі, якщо в майбутньому до класу будуть додані інші методи, які не потребують синхронізації.

Питання

1. Що таке блокуючі черги в Java та яку проблему багатопотокового програмування вони розв’язують?
2. Які основні відмінності між інтерфейсами BlockingQueue, TransferQueue та BlockingDeque?
3. У чому полягає концепція блокування при роботі з чергами та деками?
4. Які методи додавання елементів у BlockingQueue існують і чим вони відрізняються (add, offer, put)?
5. Які методи отримання елементів із блокуючої черги використовуються та в яких випадках вони блокують потік (poll, take, peek)?
6. Як працюють тайм-аути при виклику методів offer і poll у блокуючих чергах?
7. Які особливості реалізації та використання класу ArrayBlockingQueue?
8. Для яких задач призначена DelayQueue та які вимоги висуваються до елементів, що в неї додаються?
9. Чим відрізняється LinkedBlockingQueue від ArrayBlockingQueue з точки зору структури та продуктивності?
10. Які особливості має PriorityBlockingQueue та як у ній визначається порядок обробки елементів?
11. У чому специфіка роботи SynchronousQueue і чому вона не зберігає елементи?
12. Яке призначення інтерфейсу TransferQueue і як працюють методи transfer() та tryTransfer()?
13. Які можливості надає LinkedTransferQueue порівняно зі звичайними блокуючими чергами?
14. Чим BlockingDeque відрізняється від BlockingQueue та які операції дозволяють працювати з обох кінців структури даних?
15. Які сценарії використання є типовими для LinkedBlockingDeque у багатопотокових додатках?