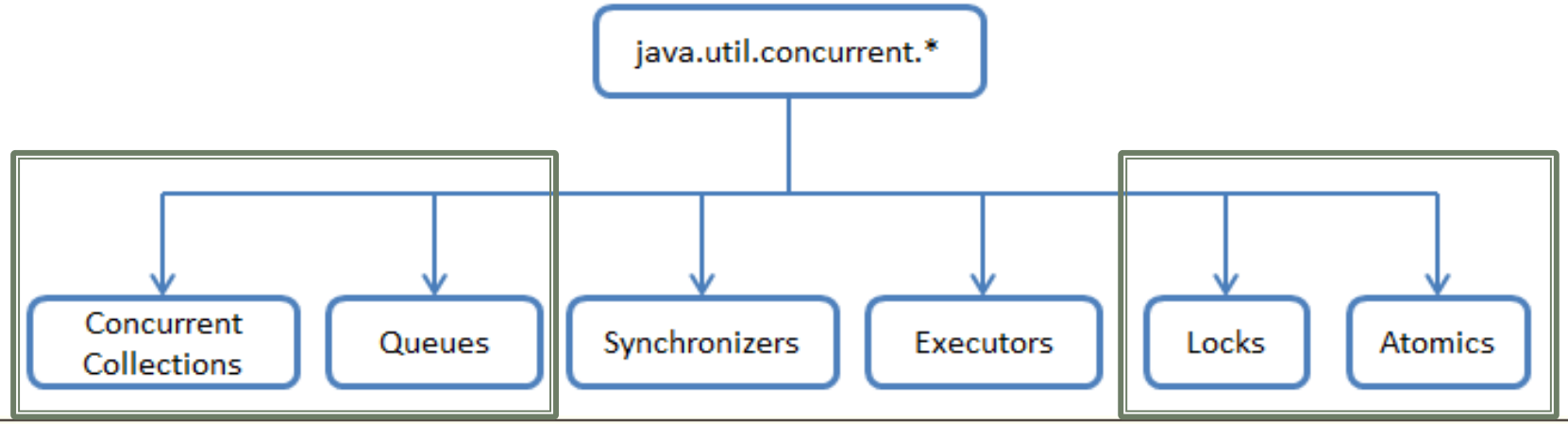

ОГЛЯД ІНШИХ ЕЛЕМЕНТІВ CONCURRENCY UTILITIES FRAMEWORK

Питання 6.5.

Проблеми потокобезпечних колекцій

- Класи Collections Framework не є потокобезпечними (thread-safe), проте їх можна зробити такими, якщо використати ***синхронізовані методи-обгортки (synchronized wrapper method)*** з класу java.util.Collections.
  - Наприклад, можна передати екземпляр ArrayList в Collections.synchronizedList() для отримання потокобезпечного варіанту ArrayList.
- Хоч вони спрощують код в багатопоточному середовищі, існує кілька проблем з thread-safe колекціями:
  - Необхідно отримати замок (acquire a lock) до ітерування по колекції, яку може змінити інший потік під час свого проходу.
    - Якщо замок не здобуто, а колекція змінена, дуже ймовірно, що буде викинуто java.util.ConcurrentModificationException.
    - Класи Collections Framework повертають *fail-fast ітератори*, які є ітераторами, що викидають ConcurrentModificationException, коли колекції змінюються в процесі проходу по ним.
    - Fail-fast ітератори часто не підходять для багатопоточних додатків.
  - Продуктивність знижується при частому зверненні до синхронізованих колекцій з багатьох потоків.
    - Ця проблема продуктивності в кінцевому рахунку впливає на масштабованість (*scalability*) додатку.

- Concurrency Utilities framework вирішує ці проблеми, вводячи ефективні та високомасштабовані collections-oriented типи.
- Їх collections-oriented класи повертають *weakly consistent imepamopu* – ітератори з наступними властивостями:
  - Елемент, що було видалено після початку ітерування, проте ще не було повернено за допомогою методу next() ітератора, не буде повертатись.
  - Елемент, що додано після початку ітерування може повертатись або не повертатись.
  - Жоден елемент не повертається більше одного разу в процесі проходження по колекції, незалежно від здійснених змін в колекції протягом ітерування.

- Короткий список concurrency-oriented collection types, що знаходяться в пакеті java.util.concurrent:
  - BlockingQueue – підінтерфейс java.util.Queue, який також підтримує блокуючі операції, які очікують,
    - поки черга стане непорожньою до отримування елементу з неї
    - поки з’явиться місце в черзі перед додаванням елементу в неї.
    - Класи ArrayBlockingQueue, DelayQueue, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue реалізують цей інтерфейс.
  - ConcurrentMap – підінтерфейс java.util.Map, що оголошує додаткові атомістичні методи putIfAbsent(), remove() та replace().
    - Класи ConcurrentHashMap (багатопоточна версія java.util.HashMap), ConcurrentNavigableMap, та ConcurrentSkipListMap реалізують його.

Демонстрація BlockingQueue та ArrayBlockingQueue

```
public class PC
{
   public static void main(String[] args)
   {
      final BlockingQueue<Character> bq;
      bq = new ArrayBlockingQueue<Character>(26);
      final ExecutorService executor = Executors.newFixedThreadPool(2);
      Runnable producer;
      producer = new Runnable()
                 {
                    @Override
                    public void run()
                    {
                       for (char ch = 'A'; ch <= 'Z'; ch++)
                       {
                          try
                          {
                             bq.put(ch);
                             System.out.println(ch + " produced by producer.");
                          }
                          catch (InterruptedException ie)
                          {
                             assert false;
                          }
                       }
                    }
                 };
      executor.execute(producer);
      Runnable consumer;
      consumer = new Runnable()
                 {
                    @Override
                    public void run()
                    {
                       char ch = '\0';
                       do
                       {
                          try
                          {
                             ch = bq.take();
                             System.out.println(ch + " consumed by consumer.");
                          }
                          catch (InterruptedException ie)
                          {
                             assert false;
                          }
                       }
                       while (ch != 'Z');
                       executor.shutdownNow();
                    }
                 };
      executor.execute(consumer);
   }
}
```

- Приклад producer-consumer додатку на основі BlockingQueue набагато простіший, оскільки не має справу з синхронізацією.
  - Використовуються методи put() та take() з BlockingQueue, щоб додати та видалити об’єкт відповідно з блокуючої черги.
  - put() блокує додавання, коли немає місця; take() блокує видалення з порожньої черги.

```
"C:\Program Files\Java\j
A produced by producer.
A consumed by consumer.
B produced by producer.
C produced by producer.
D produced by producer.
E produced by producer.
F produced by producer.
G produced by producer.
H produced by producer.
I produced by producer.
J produced by producer.
K produced by producer.
B consumed by consumer.
C consumed by consumer.
L produced by producer.
D consumed by consumer.
M produced by producer.
E consumed by consumer.
N produced by producer.
F consumed by consumer.
O produced by producer.
G consumed by consumer.
H consumed by consumer.
I consumed by consumer.
J consumed by consumer.
K consumed by consumer.
L consumed by consumer.
M consumed by consumer.
N consumed by consumer.
O consumed by consumer.
P consumed by consumer.
P produced by producer.
Q produced by producer.
R produced by producer.
S produced by producer.
T produced by producer.
```

Клас ConcurrentHashMap

- Даний клас поводить себе так, як і HashMap, проте він спроектований для роботи в багатопоточному середовищі без потреби явної синхронізації.
- Наприклад, часто потрібно перевіряти, чи містить карта конкретне значення, і якщо його немає, додати його в карту.
  - ```
    if (!map.containsKey("some string-based key"))
      map.put("some string-based key", "some string-
                          based value");
    ```
  - Цей код не є потокобезпечним.
  - Між викликом map.containsKey() та map.put() може запуститись та вставити власне значення інший потік.
  - Тоді це значення може перезаписатись.
    ```

Вирішення проблеми - I

- Потрібно явно синхронізувати код:

```
synchronized(map)
{
   if (!map.containsKey("some string-based key"))
      map.put("some string-based key", "some string-based value");
}
```

  - Проблема з таким підходом: весь меп замикається для операцій зчитування та запису, поки йде перевірка на існування ключа та додавання entry в меп, якщо ключ не існує.
  - блокування (locking) впливає на продуктивність, коли багато потоків намаагаються отримати доступ до мепу.

Вирішення проблеми – II

- Узагальнений клас ConcurrentHashMap<V> вирішує цю проблему, постачаючи метод V putIfAbsent(K key, V value), який представляє key/value entry в меп, якщо ключ відсутній.
  - Метод еквівалентний фрагменту коду, проте має кращу продуктивність:

```
synchronized(map)
{
   if (!map.containsKey(key))
      return map.put(key, value);
   else
      return map.get(key);
}
```

  - Використовуючи putIfAbsent(), попередній код перетворюється в простіший фрагмент:
  - map.putIfAbsent("some string-based key", "some string-based value");

Атомарні змінні. Пакет java.util.concurrent.atomic

- Внутрішні замки, що пов’язані з моніторами об’єктів, історично мають погану продуктивність.
  - Хоч вона з часом покращилась, все ще присутній bottleneck при створенні веб-серверів та додатків, які вимагають високої масштабованості та продуктивності за наявності значної неузгодженості потоків (thread contention).
- Багато досліджень було проведено для розробки неблокуючих алгоритмів, які можуть радикально покращити продуктивність в контексті синхронізації.
  - Ці алгоритми пропонують покращену масштабованість, оскільки потоки не блокуються, коли кілька потоків сперечаються за одні й ті ж дані.
  - Також потоки не страждають від дедлоків та інших проблем liveness problems.
  - Java 5 забезпечила можливість створювати ефективні неблокуючі алгоритми, представивши пакет java.util.concurrent.atomic.
  - Згідно з документацією, пакет постачає невеликий toolkit класів, що підтримують lock-free, thread-safe операції над єдиними змінними.
- Класи з пакету java.util.concurrent.atomic розширюють поняття volatile-значень, полів та елементів масиву, включаючи atomic conditional update, щоб зовнішня синхронізація не була потрібною.

Деякі з класів з java.util.concurrent.atomic

- ***AtomicBoolean:*** boolean-значення, яке можна атомарно оновити.
- ***AtomicInteger:*** int-значення, яке можна атомарно оновити.
- ***AtomicIntegerArray:*** int-масив, чиї елементи можна атомарно оновити.
- ***AtomicLong:*** long-значення, яке можна атомарно оновити.
- ***AtomicLongArray:*** long-масив, чиї елементи можна оновити атомарно.
- ***AtomicReference:*** посилання на об’єкт, який можна оновити атомарно.
- ***AtomicReferenceArray:*** посилання на масив, чиї елементи можна оновити атомарно.
  - Атомарні змінні використовуються для реалізації лічильників, генераторів послідовностей (наприклад, java.util.concurrent.ThreadLocalRandom) та інших конструктів, які вимагають взаємного виключення без проблем з продуктивністю при високій неузгодженості потоків

Потокобезпечне повернення унікальних ідентифікаторів

```
class ID
{
   private static volatile long nextID = 1;

   static synchronized long getNextID()
   {
      return nextID++;
   }
}
```

Замінимо на:

```
import java.util.concurrent.atomic.AtomicLong;

class ID
{
   private static AtomicLong nextID = new AtomicLong(1);

   static long getNextID()
   {
      return nextID.getAndIncrement();
   }
}
```

- Хоч код нормально синхронізовано (включаючи visibility), внутрішній замок, прив’язаний до synchronized, може погіршити продуктивність при високій конкурентності потоків.
  - Пакет java.util.concurrent.atomic включає класи DoubleAccumulator, DoubleAdder, LongAccumulator та LongAdder, що відносяться до проблеми масштабованості в контексті підтримки єдиного лічильника, суми чи іншого значення з можливістю оновлень від багатьох потоків.
  - Ці нові класи “зсередини задіюють contention-reduction підходи, що постачає величезне покращення пропускної здатності в порівнянні з атомарними змінними. Це було здійснено шляхом зменшення гарантій атомарності до прийнятного рівня в більшості додатків.”

Розуміння магії Atomic

- Низькорівневий механізм синхронізації в Java, що веде до взаємного виключення та видимості (*visibility*), впливає на завантаженість «заліза» та масштабованість:
  - *Contended synchronization* (кілька потоків постійно сперечаються за замок) – дорога, у результаті страждає пропускна здатність.
    - Дорожнеча пояснюється, в першу чергу, частим переключенням контексту (*context switching).*
    - Кожне переключення контексту може забирати багато процесорних циклів для виконання.
    - Сучасні віртуальні машини Java використовують недорогу *uncontended synchronization*.
  - Коли потік, що тримає замок, відкладає роботу (наприклад, через планувальника), жоден з потоків, що потребують замок, не прогресує; апаратне забезпечення isn't utilized так, як могло б.
- Використання volatile як альтернативи синхронізації не є вирішенням.
  - Volatile-змінні вирішують лише проблему видимості.
  - Вони не можуть безпечно реалізувати атомарні read-modify-write послідовності, що необхідні для реалізації потокобезпечних лічильників та інших сутностей, які потребують взаємного виключення.
  - Проте існує альтернатива, яка відповідає за переваги в продуктивності в concurrency utilities (наприклад, класі java.util.concurrent.Semaphore).

Альтернатива: Compare-and-swap (CAS)

- Загальний термін для специфічних для мікропроцесора інструкцій, що не перериваються та зчитують місце в пам’яті, порівнюють зчитане та очікуване значення і зберігають нове значення в даному місці, коли зчитане та очікуване значення співпадають.
  - Інакше, нічого не робиться.
  - Сучасні процесори пропонують варіації CAS. Наприклад, процесори Intel постачають сімейство інструкцій cmpxchg, в той час, як старіші PowerPC-процесори постачають еквівалент – load-link (зокрема, lwarx) та store-conditional (як stwcx) інструкції.
- Зазвичай CAS використовують так:
  1) Зчитується значення *x* за адресою *A*.
  2) Виконується багатокрокове обчислення над *x,* щоб отримати нове значення – *y*.
  3) Використовується CAS, щоб змінити значення в комірці *A* з *x* на *y*. CAS виконується успішно, коли значення в *A* не змінилось, виконуючи ці кроки.
- Для розуміння переваг CAS розглянемо лістинг класу ID, який повертає унікальний ідентифікатор.
  - Оскільки клас оголошує свій метод getNextID() синхронізованим, висока конкуренція за замок монітора призводить до надмірного перемикання контексту, що може затримати всі потоки та призвести до поганої масштабованості додатку.

- Припустимо існування класу CAS, що зберігає цілочисельне значення у value.
  - Крім того, він пропонує атомарні методи int getValue() для повернення value та int compareAndSwap(int expectedValue, int newValue) для реалізації CAS.
  - За кулісами CAS покладається на Java Native Interface [JNI], щоб отримати доступ до специфічних для мікропроцесора CAS-інструкцій.
  - Метод compareAndSwap() виконує наступну послідовність інструкцій атомарно:

```
int readValue = value;              // Obtain the stored value.
if (readValue == expectedValue)     // If stored value not modified ...
   value = newValue;                // ... change to new value.
return readValue;                   // Return value before a potential change.
```

Нова версія класу ID, яка використовує клас CAS для отримання унікального id у високопродуктивний спосіб

```
class ID
{
   private static CAS value = new CAS(1);

   static long getNextID()
   {
      int curValue = value.getValue();
      while (value.compareAndSwap(curValue, curValue + 1) != curValue)
         curValue = value.getValue();
      return curValue - 1;
   }
}
```

- ID інкапсулює екземпляр CAS, який ініціалізується 1 (ціле число) та оголошує метод getNextID() для отримання поточного значення ідентифікатора, а потім – його інкременту.
  - Після одержання поточного значення екземпляру, getNextID() повторно викликає compareAndSwap(), поки значення curValue не змінилось (іншим потоком).
  - Тоді метод може змінити це значення, після чого повертає попереднє значення.
  - Коли не задіяно замка, конкурентний доступ та надмірне перемикання контексту уникаються.
  - Продуктивність покращується, а код стає більш масштабованим.

Приклад покращень concurrency utilities від CAS

- Розглянемо java.util.concurrent.locks.ReentrantLock.
  - Клас пропонує кращу продуктивність, ніж synchronized при високій конкурентності потоків.
  - Для підвищення продуктивності синхронізація ReentrantLock управляється підкласом абстрактного класу java.util.concurrent.locks.AbstractQueuedSynchronizer.
- У свою чергу, цей клас отримує переваги від недокументованого класу sun.misc.Unsafe та його CAS-методу compareAndSwapInt().
  - Класи атомарних змінних також отримують переваги від CAS.
  - Крім того, вони постачають метод
    boolean compareAndSet(expectedValue, updateValue)
  - Він (з різними аргументами залежно від класу) автоматично задає змінній значення updateValue, коли на поточний момент вона має значення expectedValue, повертаючи true при успіху.
- Пакет java.util.concurrent.locks забезпечує бібліотеку інтерфейсів та класів для блокування (locking) та очікування for conditions у стилі, відмінному від вбудованого механізму синхронізації та wait/notification для java.lang.Object.
  - Locking Framework покращує синхронізацію та wait/notification, пропонуючи такі можливості, як lock polling та timed waits.

Інтерфейс Lock

- Пропонує більш екстенсивні операції блокування (locking operations), ніж замки з моніторами.
  - Наприклад, можна негайно відхилити спробу отримання замка, коли замок недоступний.
- Інтерфейс оголошує методи:
  - **void lock():** отримує замок. Коли замок недоступний, викликаючий потік змушений чекати, поки замок не стане доступним.
  - **void lockInterruptibly():** отримує замок, якщо викликаючий потік не перервано. Коли замок недоступний, викликаючий потік змушений чекати, поки замок не стане доступним або потік не буде перервано, що призведе до викидання InterruptedException.
  - **Condition newCondition():** повертає новий екземпляр Condition, прив'язаний до екземпляру Lock. Метод викидає java.lang.UnsupportedOperationException, коли реалізація Lock не підтримує умови (conditions).
  - **boolean tryLock():** отримує замок, коли він доступний в момент виклику цього методу. Повертає true, коли замок отримано та false в іншому випадку.
  - **boolean tryLock(long time, TimeUnit unit):** отримує замок, коли він доступний в момент заданого проміжку часу, а викликаючий потік не перервано. Коли замок недоступний, викликаючий потік змушений чекати, поки замок стане доступнимпротягом деякого часу очікування або потік буде перервано, що призводить до викидання InterruptedException.
  - **void unlock():** звільняє замок.

- Отримані замки мають звільнятись.
  - В контексті synchronized-методів та виразів неявний замок монітора, пов’язаний з кожним об’єктом, всі отримання та звільнення замка відбуваються в block-structured стилі.
  - Коли кілька замків отримуються, вони вивільняються у зворотному порядку, а всі замки звільняються в тому ж lexical scope, в якому вони отримувались.
- Отримання та звільнення замка в контексті реалізацій інтерфейсу Lock може бути більш гнучким.
  - Наприклад, деякі алгоритми обходу виконують конкурентний доступ до структур даних, що вимагає використання hand-over-hand або chain locking: отримує замок вузла A, потім вузла B, потім звільняємо A та отримуємо C, потім звільняємо B і отримуємо D і т. д.
  - Реалізації інтерфейсу Lock дають можливість використовувати такий підхід, дозволяючи замку отримуватись та звільнятись у різних scopes, а також дозволяючи кільком замкам отримуватись та звільнятись в довільному порядку.

- Відсутність block-structured locking видаляє автоматичне звільнення замків, що відбувається з synchronized-методами та виразами.
  - У результаті зазвичай задіюють наступний код для отримання та звільнення замка:

```
Lock l = ...; // ... is a placeholder for code that obtains the lock
l.lock();
try
{
  // access the resource protected by this lock
}
catch (Exception ex)
{
  // restore invariants
}
finally
{
   l.unlock();
}
```

Цей шаблон коду обов’язкове звільнення будь-якого отриманого замка.

Note Від усіх реалізацій Lock вимагають задіювання тієї ж семантики синхронізації пам’яті, що постачається вбудованим monitor lock.

Клас ReentrantLock

- Інтерфейс Lock реалізується за допомогою класу ReentrantLock, який описує взаємовиключне блокування з повторним входом (reentrant mutual exclusion lock).
  - Цей замок пов’язаний з hold count.
  - Коли потік тримає замок та заново отримує (reacquires) замок за допомогою lock(), lockUninterruptibly() чи одного з методів tryLock(), hold count збільшується на 1.
  - Коли потік викликає unlock(), hold count зменшується на 1.
  - Замок звільняється, коли count досягає 0.
- ReentrantLock пропонує ту ж семантику роботи з багатопоточністю та пам’яттю, як і неявний monitor lock, доступ до якого одержується за допомогою synchronized-методів та виразів.
  - Проте він має розширені можливості та пропонує кращу продуктивність при високій конкуренції потоків (потоки часто звертаються за отриманням замка, який уже тримається іншим потоком).
  - Коли багато потоків намагаються отримати доступ до спільного ресурсу, JVM проводить менше часу за плануванням цих потоків, а більше – за їх виконанням.

Клас ReentrantLock

- Конструктори ReentrantLock:
  - **ReentrantLock():** створює екземпляр ReentrantLock. Еквівалентний конструктору ReentrantLock(false).
  - **ReentrantLock(boolean fair):** створює екземпляр ReentrantLock із заданою fairness policy. При fair=true замок має використовувати fair ordering policy: при конкуренції замок буде віддавати перевагу надаванню доступу потоку, що очікує найдовше.
- ReentrantLock реалізує методи Lock.
  - Проте його реалізація unlock() викидає java.lang.IllegalMonitorStateException, коли викликаючий потік не буде тримати замок.
- Також ReentrantLock постачає свої власні методи.
  - Наприклад, boolean isFair() повертає fairness policy
  - boolean isHeldByCurrentThread() повертає true, коли замок is held поточним потоком (current thread).

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class PC
{
   public static void main(String[] args)
   {
      final Lock lock = new ReentrantLock();
      final BlockingQueue<Character> bq;
      bq = new ArrayBlockingQueue<Character>(26);
      final ExecutorService executor = Executors.newFixedThreadPool(2);
      Runnable producer;
      producer = new Runnable()
                {
                   @Override
                   public void run()
                   {
                      for (char ch = 'A'; ch <= 'Z'; ch++)
                      {
                         try
                         {
                            lock.lock();
                            try
                            {
                               while (!bq.offer(ch))
                               {
                                  lock.unlock();
                                  Thread.sleep(50);
                                  lock.lock();
                               }
                               System.out.println(ch + " produced by producer.");
                            }
                            catch (InterruptedException ie)
                            {
                               assert false;
                            }
                         }
                         finally
                         {
                            lock.unlock();
                         }
                      }
                   }
                };
```

Досягення синхронізації в термінології замків

- Лістинг демонструє Lock та ReentrantLock у варіації попереднього лістингу, що завжди забезпечує вивід у коректному порядку
  - consumed message appearing before a produced message.

```
executor.execute(producer);
Runnable consumer;
consumer = new Runnable()
           {
              @Override
              public void run()
              {
                 char ch = '\0';
                 do
                 {
                    try
                    {
                       lock.lock();
                       try
                       {
                               Character c;
                               while ((c = bq.poll()) == null)
                               {
                                  lock.unlock();
                                  Thread.sleep(50);
                                  lock.lock();
                               }
                               ch = c; // unboxing behind the scenes
                               System.out.println(ch + " consumed by consumer.");
                            }
                            catch (InterruptedException ie)
                            {
                               assert false;
                            }
                         }
                         finally
                         {
                            lock.unlock();
                         }
                      }
                      while (ch != 'Z');
                      executor.shutdownNow();
                   }
                };
        executor.execute(consumer);
    }
 }
```

- Лістинг використовує методи lock() та unlock() з інтерфейсу Lock, щоб отримати та звільнити замок.
  - Коли потік викликає lock(), а замок недоступний, потік is disabled (і не входить у планування), поки замок не стане доступним.
- Лістинг також використовує метод offer() з класу BlockingQueue замість put(), щоб зберігати об’єкт у блокуючій черзі, та метод poll() замість take(), щоб отримати об’єкт з черги.
  - Ці альтернативні методи використовуються через те, що вони не блокують.
- При запуску додатку можна помітити, що він ніколи не виводить consuming message до producing message для одного і того ж елементу.

Сценарій дедлоку за умови використання put() і take()

1) Потік-споживач отримує замок за допомогою виклику lock.lock().
2) Потік-виробник намагається отримати замок за допомогою lock.lock() та is disabled, оскільки замок уже тримає потік-споживач.
3) Потік-споживач викликає take(), щоб отримати наступний об’єкт java.lang.Character з черги.
4) Оскільки черга порожня, потік-споживач повинен чекати.
5) Потік-споживач не віддає замок, який вимагає потік-виробник, який теж продовжує чекати.

Інтерфейс Condition

- Виносить (factors out) методи wait(), notify() та notifyAll() класу Object в окремі condition-об’єкти, щоб to give the effect наявності кількох wait-sets на об’єкт per object, комбінуючи їх за допомогою довільних реалізацій інтерфейсу Lock.
  - Там, де Lock заміняє synchronized-методи та вирази, Condition заміняє методи wait()/notify()/notifyAll().
- **Note** Екземпляр Condition має вбудовану прив’язку до замка.
  - Щоб отримати екземпляр Condition для конкретного екземпляру Lock, використовуйте метод newCondition() з Lock.
- Методи інтерфейсу:
  - **void await()**: змушує викликаючий потік чекати, поки його не розбудять (signal) чи перервуть (interrupt).
  - **boolean await(long time, TimeUnit unit)**
  - **long awaitNanos(long nanosTimeout)**: змушує викликаючий потік чекати, поки його не розбудять (signal), перервуть (interrupt) чи не пройде заданий період часу.
  - **void awaitUninterruptibly()**: змушує викликаючий потік чекати, поки його не розбудять (until it’s signaled).
  - **boolean awaitUntil(Date deadline)**: змушує викликаючий потік чекати, поки його не розбудять (signal), перервуть (interrupt) чи не пройде заданий deadline.
  - **void signal()**: «будить» один чекаючий потік.
  - **void signalAll()**: «будить» усі чекаючі потоки.

Досягнення синхронізації з точки зору Locks та Conditions

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class PC
{
   public static void main(String[] args)
   {
      Shared s = new Shared();
      new Producer(s).start();
      new Consumer(s).start();
   }
}
```

```
class Shared
{
   private char c;

   private volatile boolean available;

   private final Lock lock;

   private final Condition condition;

   Shared()
   {
      available = false;
      lock = new ReentrantLock();
      condition = lock.newCondition();
   }
```

- Метод main() інстанціює класи Shared, Producer та Consumer.
  - Екземпляр Shared передається в конструктори Producer та Consumer, а потім ці потоки розпочинають роботу.

Продовження коду (клас Shared)

```
Lock getLock()
{
   return lock;
}

char getSharedChar()
{
   lock.lock();
   try
   {
      while (!available)
         try
         {
            condition.await();
         }
         catch (InterruptedException ie)
         {
            ie.printStackTrace();
         }
      available = false;
      condition.signal();
   }
   finally
   {
      lock.unlock();
      return c;
   }
}
```

```
void setSharedChar(char c)
{
   lock.lock();
   try
   {
      while (available)
         try
         {
            condition.await();
         }
         catch (InterruptedException ie)
         {
            ie.printStackTrace();
         }
      this.c = c;
      available = true;
      condition.signal();
   }
   finally
   {
      lock.unlock();
   }
}
```

```
class Producer extends Thread
{
   private final Lock l;

   private final Shared s;

   Producer(Shared s)
   {
      this.s = s;
      l = s.getLock();
   }

   @Override
   public void run()
   {
      for (char ch = 'A'; ch <= 'Z'; ch++)
      {
         l.lock();
         s.setSharedChar(ch);
         System.out.println(ch + " produced by producer.");
         l.unlock();
      }
   }
}
```

```
class Consumer extends Thread
{
   private final Lock l;

   private final Shared s;

   Consumer(Shared s)
   {
      this.s = s;
      l = s.getLock();
   }

   @Override
   public void run()
   {
      char ch;
      do
      {
         l.lock();
         ch = s.getSharedChar();
         System.out.println(ch + " consumed by consumer.");
         l.unlock();
      }
      while (ch != 'Z');
   }
}
```

Розглянемо конструктор Shared

- У ньому створюється замок (lock = new ReentrantLock();) та condition, пов’язана з цим замком (condition = lock.newCondition();).
  - Цей замок стає доступним потоку-виробнику та потоку-споживачу за допомогою методу getLock() з Lock.
- Потік-виробник викликає метод void setSharedChar(char c) з класу Shared, щоб згенерувати новий символ, а потім вивести повідомлення, що інформує про вироблений символ.
  - Цей метод замикає попередньо створений об’єкт Lock та входить у цикл while, що повторно перевіряє змінну available, яка має значення true, коли вироблений символ доступний для споживання.
  - Поки available = true, виробник викликає метод condition’s await(), щоб чекати, поки available не набуде значення false.
  - Споживач signals the condition, щоб «розбудити» виробника, коли символ буде спожито.
- Після виходу з циклу потік-виробник записує новий символ, присвоює available значення true, щоб вказати на доступність символу для споживання, and signals the condition «розбудити» очікуючого споживача.
  - У кінці він відмикає (unlock) замок та виходить з setSharedChar()

Зауважте

- Блок setSharedChar()/System.out.println() замкнуто (lock) в методі run() класу Producer, а блок getSharedChar()/ System.out.println() – в методі run() класу Consumer, щоб уникнути виводу додатком consuming messages до producing messages, навіть незважаючи на те, що символи вироблені до свого споживання.
  - Поведінка потоку-споживача і методу getSharedChar() аналогічні поведінці потоку-виробнику та методу setSharedChar().
- Шаблон try/finally для забезпечення видалення (dispose) замку в методі run() з класів Producer та Consumer не використовувався, оскільки з цього контексту не викидається виключення.

Вивід програми

- вказує на *lockstep synchronization*
  - потік-виробник не виробляє елемент, доки старий не буде спожито, а потік-споживач не споживає ще не вироблений елемент

- ```
  A produced by producer.
  A consumed by consumer.
  B produced by producer.
  B consumed by consumer.
  C produced by producer.
  C consumed by consumer.
  D produced by producer.
  D consumed by consumer.
  ```
  ```

ReadWriteLock

- Виникають ситуації, коли структури даних читаються частіше, ніж змінюються.
  - Наприклад, створено онлайн-словник визначень, який буде читати багато потоків у паралельному режимі, а окремий потік буде час від часу додавати нові визначення чи оновлювати існуючі.
  - Locking Framework постачає замикаючий механізм зчитування/запису (readwrite locking mechanism) для таких ситуацій, коли при зчитуванні важлива висока конкуренція, а при запису – безпека ексклюзивного доступу.
  - Цей механізм базується на інтерфейсі ReadWriteLock.
- ReadWriteLock підтримує пару замків: на read-only операції та операції запису.
  - Замок на зчитування може be held одночасно кількома потоками-читачами, доки немає потоків-записувачів.
  - Замок на запис ексклюзивний: лише один потік може змінювати спільні (shared) дані (замок, пов’язаний з ключовим словом synchronized також ексклюзивний).
- ReadWriteLock оголошує методи, що повертають ці замки
  - Lock readLock()
  - Lock writeLock()

ReadWriteLock реалізується класом ReentrantReadWriteLock

- Описує reentrant read-write lock зі схожою семантикою до ReentrantLock.
- Конструктори ReentrantReadWriteLock:
  - **ReentrantReadWriteLock():** створює екземпляр ReentrantReadWriteLock, еквівалентний ReentrantReadWriteLock(false).
  - **ReentrantReadWriteLock(boolean fair):** створює екземпляр ReentrantReadWriteLock із заданою fairness policy. Pass true to fair when this lock should use a fair ordering policy.
- **Note Для** fair ordering policy, коли поточний утриманий замок звільняється, either the longest-waiting single writer thread will be assigned the write lock or, when there's a group of reader threads waiting longer than all waiting writer threads, that group will be assigned the read lock.
- Потік, що намагається отримати fair read lock (non-reentrantly), буде блокуватись, коли write lock is held або присутній очікуючий writer-потік.
- Потік не буде отримувати read lock, поки after the oldest currently waiting writer thread has acquired and released the write lock.
  - Якщо очікуючий writer abandons its wait, leaving one or more reader threads as the longest waiters in the queue with the write lock free, those readers will be assigned the read lock.
- Потік, що намагається отримати fair write lock (non-reentrantly) will block, поки обидва (read lock and write lock) замки не будуть вільними (тобто, не буде очікуючих потоків).
  - Неблокуючі методи tryLock() не беруть до уваги fair setting і за можливості будуть негайно отримувати замок, незалежно від очікуючих потоків.

- Після інстанціювання цього класу для отримання замків на зчитування та запис викликають методи:
  - ReentrantReadWriteLock.ReadLock readLock().
  - ReentrantReadWriteLock.WriteLock writeLock().
- Кожен із вкладених класів ReadLock та WriteLock реалізує інтерфейс Lock та оголошує власні методи.
- Крім того, ReentrantReadWriteLock оголошує додаткові методи:
  - **int getReadHoldCount():** повертає number of reentrant read holds on this lock by the calling thread, which is 0 when the read lock isn't held by the calling thread. A reader thread has a hold on a lock for each lock action that's not matched by an unlock action.
  - **int getWriteHoldCount():** повертає number of reentrant write holds on this lock by the calling thread, which is 0 when the write lock isn't held by the calling thread. A writer thread has a hold on a lock for each lock action that's not matched by an unlock action.

```
public class Dictionary
{
   public static void main(String[] args)
   {
      final String[] words =
      {
         "hypocalcemia",
         "prolixity",
         "assiduous",
         "indefatigable",
         "castellan"
      };

      final String[] definitions =
      {
         "a deficiency of calcium in the blood",
         "unduly prolonged or drawn out",
         "showing great care, attention, and effort",
         "able to work or continue for a lengthy time without tiring",
         "the govenor or warden of a castle or fort"
      };

      final Map<String, String> dictionary = new HashMap<String, String>();

      ReadWriteLock rwl = new ReentrantReadWriteLock(true);
      final Lock rlock = rwl.readLock();
      final Lock wlock = rwl.writeLock();
```

- Головний потік спочатку створює слова та визначення як масиви рядків,
  - Оголошуються final, оскільки доступ до них буде з анонімних класів.
  - Після створення мепу для пар word/definition, потік одержує reentrant read/write lock та отримує доступ до замків на зчитування та запис.

```
final Map<String, String> dictionary = new HashMap<String, String>();

ReadWriteLock rwl = new ReentrantReadWriteLock(true);
final Lock rlock = rwl.readLock();
final Lock wlock = rwl.writeLock();

Runnable writer = () ->
                  {
                     for (int i = 0; i < words.length; i++)
                     {
                        wlock.lock();
                        try
                        {
                           dictionary.put(words[i],
                                          definitions[i]);
                           System.out.println("writer storing " +
                                              words[i] + " entry");
                        }
                        finally
                        {
                           wlock.unlock();
                        }

                        try
                        {
                           Thread.sleep(1);
                        }
                        catch (InterruptedException ie)
                        {
                           System.err.println("writer " +
                                              "interrupted");
                        }
                     }
                  };
```

- Тепер створено runnable-об’єкт для writer-потоку.
  - Його метод run() ітерує по масиву words.
  - Кожна ітерація locks the writer lock.
  - Коли цей метод returns, writer-потік має ексклюзивний writer lock та може оновлювати меп, викликаючи його метод put().
  - Після виводу повідомлення для ідентифікації доданого слова writer-потік звільняє замок та спить 1мс, роблячи вигляд виконання іншої роботи.
  - Екзекутор, заснований на пулі потоків, використовується для виклику writer thread’s runnable.

Завершення коду та результати виводу

```
    ExecutorService es = Executors.newFixedThreadPool(1);
    es.submit(writer);

    Runnable reader = () ->
                      {
                         while (true)
                         {
                            rlock.lock();
                            try
                            {
                               int i = (int) (Math.random() *
                                              words.length);
                               System.out.println("reader accessing " +
                                                  words[i] + ": " +
                                                  dictionary.get(words[i])
                                                  + " entry");
                            }
                            finally
                            {
                               rlock.unlock();
                            }
                         }
                      };
    es = Executors.newFixedThreadPool(1);
    es.submit(reader);
  }
}
```

```
writer storing hypocalcemia entry
writer storing prolixity entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
writer storing assiduous entry
reader accessing assiduous: showing great care, attention, and effort entry
reader accessing castellan: null entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
reader accessing assiduous: showing great care, attention, and effort entry
reader accessing indefatigable: null entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
reader accessing assiduous: showing great care, attention, and effort entry
reader accessing indefatigable: null entry
reader accessing prolixity: unduly prolonged or drawn out entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
reader accessing castellan: null entry
reader accessing assiduous: showing great care, attention, and effort entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
reader accessing prolixity: unduly prolonged or drawn out entry
reader accessing assiduous: showing great care, attention, and effort entry
reader accessing castellan: null entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
reader accessing indefatigable: null entry
reader accessing castellan: null entry
reader accessing prolixity: unduly prolonged or drawn out entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
writer storing indefatigable entry
reader accessing assiduous: showing great care, attention, and effort entry
reader accessing assiduous: showing great care, attention, and effort entry
```