Лекція 6. Розширені механізми синхронізації в мові Java

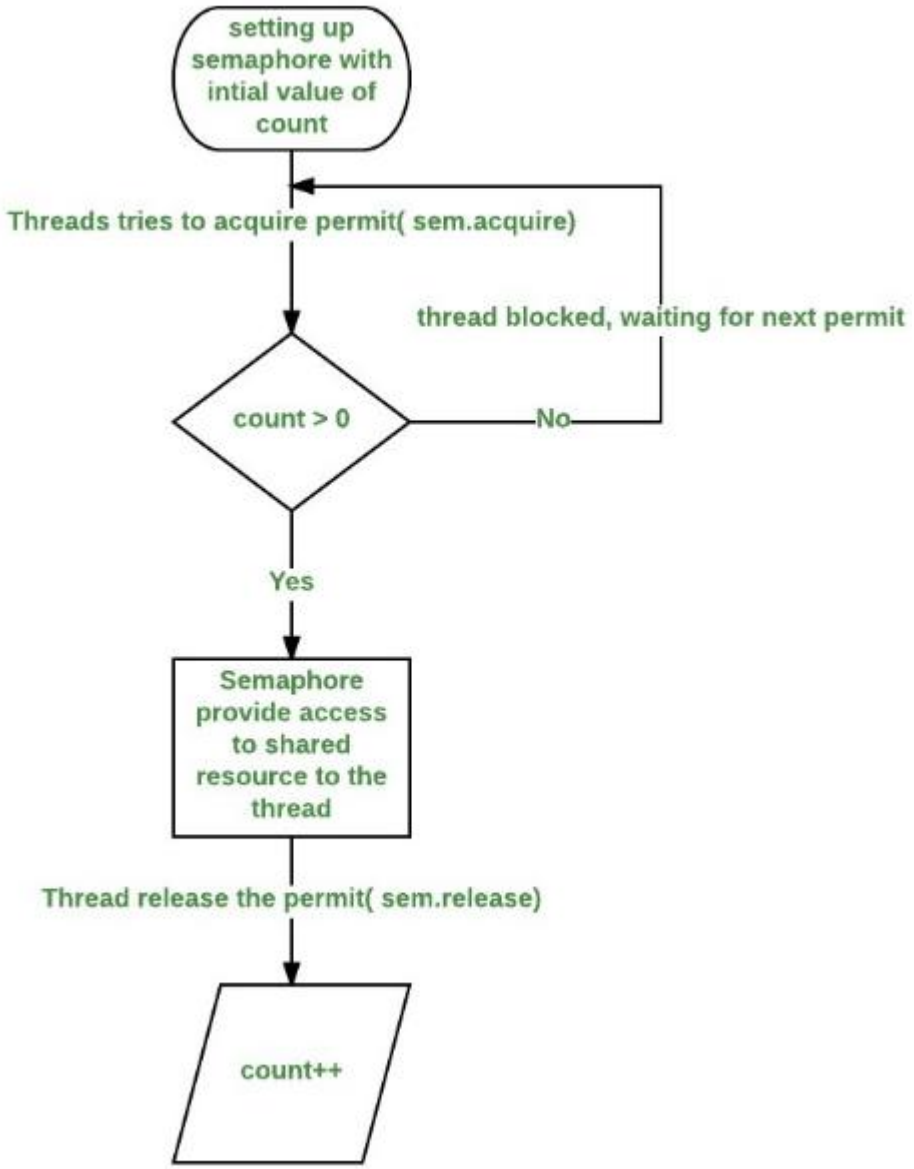
Тема 1. Блокування за допомогою лічильника. Клас Semaphore.

Тема 2. Методи класу ReentrantLock для керування взаємним виключенням, блокуванням та справедливістю доступу потоків.

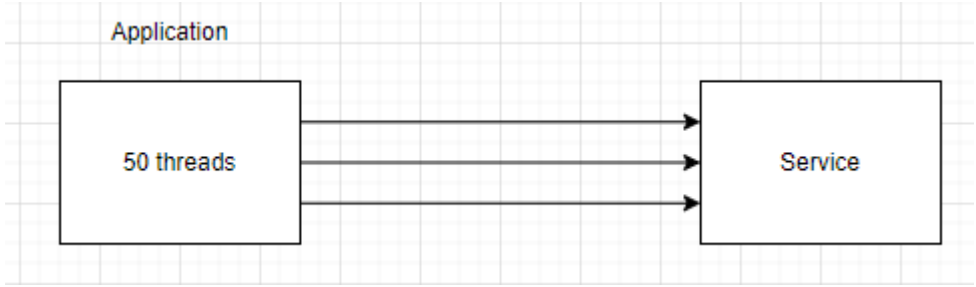
Тема 3. Методи класу ReentrantReadWriteLock та їх застосування в багатопотокових сервісах і кешах з інтенсивними операціями читання.

Semaphore - Семафор контролює доступ до загального ресурсу за допомогою лічильника, максимальне значення якого задається при ініціалізації в конструкторі.

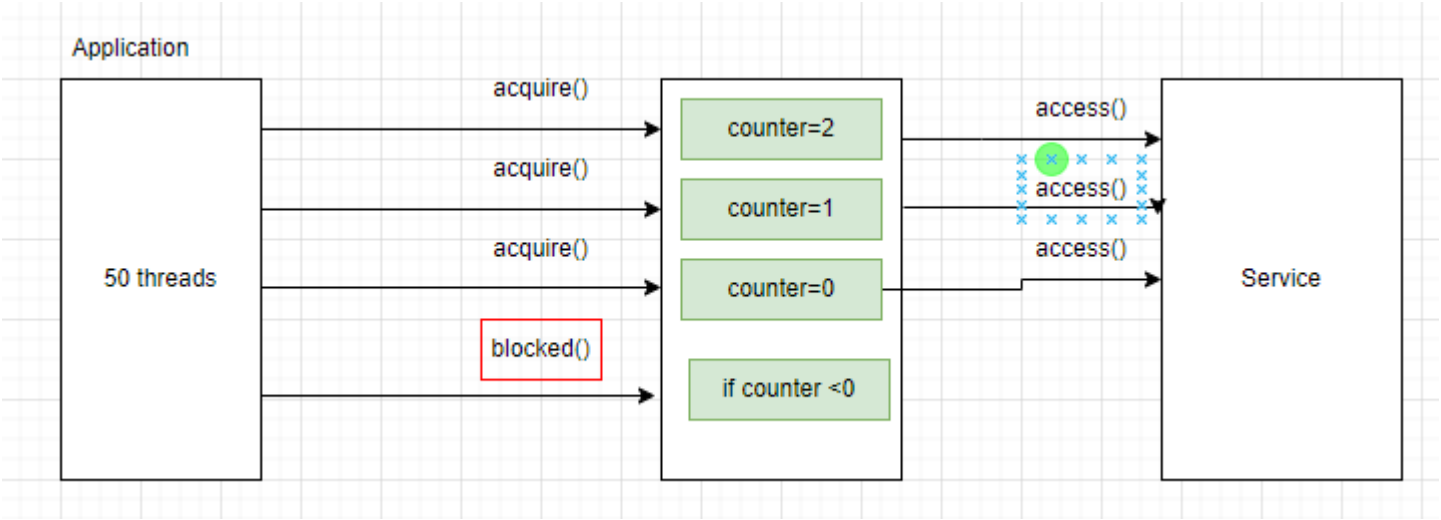
Робота семафора можна представити алгоритмом
Потік запитує у семафора доступ до ресурсу.
Семафор перевіряє лічильник, якщо його значення більше за 0 – доступ до ресурсу надається потоку і лічильних зменшується на 1.
Коли потоку більше не потрібен доступ до спільного ресурсу, тоді потік вивільняє дозвіл, що призводить до збільшення лічильника семафора.



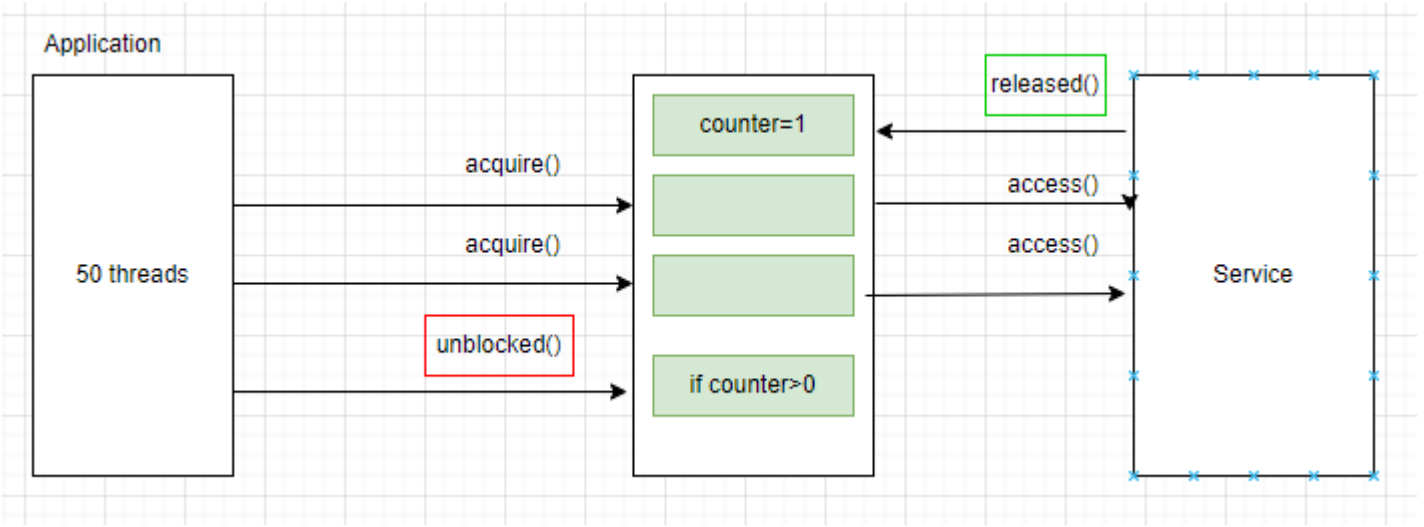
Приклад, нехай маємо «повільний» ресурс, який може обробляти дозволяє тільки одночасних запити.



В такому випадку організуємо доступ за допомогою Семафору з трьома дозволами – після надання дозволу потоку, лічильник зменшується на 1 таким чином для 4-го потоку умова лічильник більше за 0 не виконується і 4-й потік блокується.



Після, того як один з потоків повертає доступ, лічильник семафору збільшується і «заблокований потік» отримує дозвіл.



Реалізація semaphore.acquire();semaphore.release();

```
public void acquire() throws InterruptedException
{
 synchronized(this)
{
 while (permits == 0)
 { wait();
 }
 permits--;
 }
 }
```

```
public void release() {
synchronized(this)
{
permits++;
notify();
 }
 }
```

Методи класу ***Semaphore***

public Semaphore(int permits)	Конструктор класу, створює екземпляр класу Семафор , із заданою кількістю дозволів і налаштуванням «нечесної» справедливості.
public Semaphore(int permits, boolean fair)	Конструктор класу, створює екземпляр класу Семафор , із заданою кількістю дозволів і налаштуванням «чесної» справедливості.
public void acquire() throws InterruptedException	Потік, який викликає метод отримує дозвіл від Семафору на роботу з ресурсом, якщо дозвіл не отримано, то потік блокується, доки ресурс не стане вільним або потік не буде перервано.
public void acquireUninterruptibly()	На відміну від acquire(), метод acquireUninterruptibly() не викидає InterruptedException, якщо потік буде перервано під час спроби отримання дозволу. Він ігнорує переривання та продовжує намагатися отримати дозвіл. Це означає, що навіть якщо стан переривання потоку встановлено, метод не реагує, припиняючи дію з InterruptedException. Замість цього він очищує стан переривання при вході та відновлює його після повернення, якщо він був встановлений. Цей метод корисний у ситуаціях, коли критично важливо для завдання отримати дозвіл перед продовженням, і завдання не може бути передчасно завершено або пропущено через переривання. Він гарантує, що потік врешті отримає дозвіл перед тим як рухатися далі, крім випадків, коли потік вбито або система вимкнена.
public boolean tryAcquire(long timeout, TimeUnit unit) throws InterruptedException	Потік, який викликає метод отримує дозвіл від Семафору на роботу з ресурсом, при цьому очікує на доступ протягом визначеного часу. Виняткова ситуація можлива, якщо Потік перервано

Методи класу ***Semaphore***

public void release()	Потік, який викликає метод повертає ресурс, лічильник Семафора збільшується
public int availablePermits()	Кількість вільних «дозволів» на роботу з ресурсом
public int drainPermits()	Потік, який викликає метод повертає ресурс, отримує всі доступні «дозволи» Семафору
protected void reducePermits(int reduction)	Виклик методу зменшує кількість дозволів на вказане значення. Цей метод може бути корисним у підкласах, які використовують семафори для відстеження ресурсів, які стають недоступними.
protected Collection<Thread> getQueuedThreads()	Повертає колекцію потоків, які очікують в черзі Семафору на отримання «дозвіллів».
public final boolean hasQueuedThreads()	Повертає TRUE, якщо в черзі очікують потоки на доступ до ресурсу. Цей метод призначений в основному для використання в моніторингу стану системи.
public boolean isFair()	Повертає TRUE, якщо Семафор створено з налаштуванням «чесної» справедливості.

Приклад **acquire();**

```
public static class PrinterQueue {
      private final Semaphore semaphore;
public PrinterQueue() {

        // true means the semaphore will guarantee first-come, first-
served granting of permits
        int n_permits= 1;
        this.semaphore = new Semaphore(n_permits, true);
      }

      public void printJob(Object document) {
        try {
              semaphore.acquire();
          System.out.println(Thread.currentThread().getName() + ":
Printer is printing  ");
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        } finally {
          semaphore.release();
          System.out.println(Thread.currentThread().getName() +
              ": Printing done, released the printer");
       }
      }
}
```

```
public static void main(String[] args) {
   final PrinterQueue printerQueue = new PrinterQueue();
   int thread_count = 3;
   Thread[] threads = new Thread[thread_count];
   for (int i = 0; i < thread_count; i++) {
      threads[i] = new Thread(new Runnable() {
         @Override
         public void run() {
            System.out.println(Thread.currentThread().getName() +
                  ": Going to print a document");
            printerQueue.printJob(new Object());

         }
      }, "Thread " + i);
   }

for (int i = 0; i < thread_count; i++) {
      threads[i].start();
   }

for (int i = 0; i < thread_count; i++) {
      try {
         threads[i].join();
      } catch (InterruptedException e) {
         Thread.currentThread().interrupt();
      }
   }
}
```

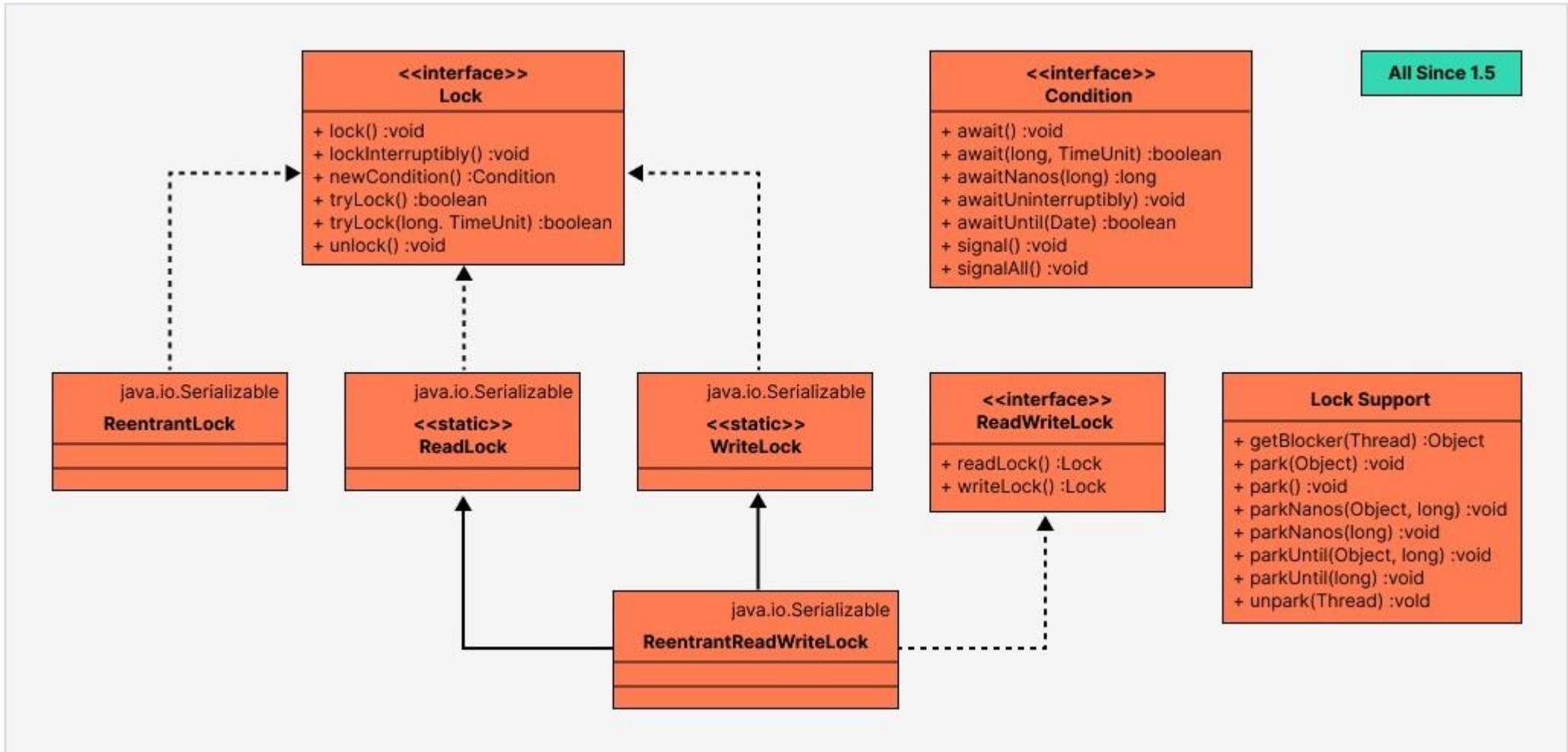
Приклад (**semaphore.tryAcquire(2, TimeUnit.*SECONDS*)**)	

ublic void printJobTryAcquire(Object document) {
try {
// Attempt to acquire the semaphore within 2 seconds
if (semaphore.tryAcquire(2, TimeUnit.*SECONDS*)) {
try {
 	
System.*out*.println(Thread.*currentThread*().getName() + ":
Printer is printing ");
 	
Thread.*sleep*(1000);
} finally {
semaphore.release();
 	
System.*out*.println(Thread.*currentThread*().getName() +
": Printing done, released the printer");
}
} else {
System.*out*.println(Thread.*currentThread*().getName()
+
": Could not acquire the printer after waiting 2
seconds, performing other tasks");
 	
performOtherTasks();
}
} catch (InterruptedException e) {
Thread.*currentThread*().interrupt();
}
}	

semaphore.acquireUninterruptibly() –зупиняємо потік!!!!

```
public void printJobacquireUninterruptibly(Object document) {
    System.out.println(Thread.currentThread().getName() + " attempting
to print a document.");
    semaphore.acquireUninterruptibly();

    try {
        System.out.println(Thread.currentThread().getName() + ": Printer
is printing  ");
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();  // Set the interrupt flag again
        System.out.println(Thread.currentThread().getName() + " was
interrupted during printing.");
    } finally {
        semaphore.release();
        System.out.println(Thread.currentThread().getName() + " has
finished printing and released the printer.");
    }
}
```

```
for (int i = 0; i < thread_count; i++) {
    threads[i].start();
    if (i == 1) {
        try {
            // Wait for thread to start and
possibly acquire the semaphore
            Thread.sleep(100);
            threads[i].interrupt();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```


<<interface>>
Lock
+ lock() :void
+ lockInterruptibly() :void
+ newCondition() :Condition
+ tryLock() :boolean
+ tryLock(long. TimeUnit) :boolean
+ unlock() :void
<<interface>>
Condition
+ await() :void
+ await(long, TimeUnit) :boolean
+ awaitNanos(long) :long
+ awaitUninterruptibly) :void
+ awaitUntil(Date) :boolean
+ signal() :void
+ signalAll() :void
All Since 1.5
java.io.Serializable
ReentrantLock
java.io.Serializable
<<static>>
ReadLock
java.io.Serializable
<<static>>
WriteLock
<<interface>>
ReadWriteLock
+ readLock() :Lock
+ writeLock() :Lock
Lock Support
+ getBlocker(Thread) :Object
+ park(Object) :void
+ park() :void
+ parkNanos(Object, long) :void
+ parkNanos(long) :void
+ parkUntil(Object, long) :void
+ parkUntil(long) :void
+ unpark(Thread) :vold
java.io.Serializable
ReentrantReadWriteLock

Lock – інтерфейс із lock framework, що надає гнучкий підхід до обмеження доступу до ресурсів/блоків у порівнянні з synchronized.
Методи інтерфейсу Lock, реалізовані в класі ReentrantLock
ReentrantLock - забезпечує можливість взаємного виключення потоків з такою самою базовою поведінкою та семантикою, що і блокування за допомогою **монітора об’єкта**, доступ до якого здійснюється за допомогою синхронізованих методів та операторів, але з розширеними можливостями.

Пригадаємо, ключове слово synchronized забезпечують синхронний доступ до спільного ресурсу для багатьох потоків методами класу Monitor. Клас Monitor є нащадком Object, тому будь-який об'єкт, визначений словом synchronized може викликати методи – wait(); notify(); notifyAll().

Методи реалізовані в класі ReentrantLock

- ✓ void lock() - запитує і отримує блокування критичної секції коду (поча-ток якої відзначено даним методом) потоком, в якому з об'єкта ReentrantLock викликається метод lock(). Якщо блокування не доступне, то поточний потік буде призупинений до тих пір, поки блокування буде звільнене іншими потоками;
- ✓ void lockInterruptibly() - запитує і отримує блокування критичної секції коду потоком, в якому з об'єкта ReentrantLock викликається даний метод, поки поточний потік не буде перерваний перериванням або блокування не буде звільнене. Якщо блокування не доступне, то поточний потік буде призупинений до тих пір, поки блокування буде звільнене іншими потоками або поки якийсь інший потік не викличе метод interrupt() з поточного потоку;
- ✓ boolean tryLock() - метод запитує і отримує блокування так само, як і lock(), але якщо блокування недоступне, потік, в якому з об'єкта ReentrantLock викликається даний метод, в стан сну не переводиться, а продовжує працювати (метод повертає true при отриманні блокування і false якщо блокування не- доступне);
- ✓ boolean tryLock(long time, TimeUnit unit) -аналогічний tryLock(), але в разі недоступності блокування потік, в якому з об'єкта. ReentrantLock викликається даний метод, засинає на зазначений як параметр час (протягом цього часу він зможе захопити блокування, яке звільниться).
- ✓ void unlock() - єдиний метод звільнення блокування критичної секції потоком, в якому з об'єкта ReentrantLock викликається даний метод.
- ✓ newCondition() – надає потокам спосіб координувати свою діяльність на основі певних умов, дозволяючи більш складні шаблони синхронізації, окрім простого отримання та зняття блокування.

ReentrantLock підтримує *справедливий* (*fair*) або *несправедливий* (*non-fair*) *доступ* до критичних секцій. При справедливому блокуванні дотримується порядок звільнення потоків, за правилом FIFO. При несправедливому розблокуванні порядок звільнення потоків не гарантується, таке розблокування працює швидше. За замовчуванням, використовується несправедливе розблокування.

Приклад, в якому методи synchronized та *lock*.lock() забезпечують однакову поведінку

```
public class CommonObject {
   private static int count =0;
   public  void  incCount() {
      synchronized (this) {
         for (int i = 0; i < 10; i++)
{

System.out.print(count++ + " ");
         }
      }
      System.out.println();
   }

   static int getCount(){
      return count;
   }
}
```

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

public class CommonObjectLock {
   private static int count =0;
   private static ReentrantLock lock = new ReentrantLock();
   public  void  incCount() {

         lock.lock();
         try{
         for (int i = 0; i < 10; i++) {
            System.out.print(count++ + " ");
         }
         } finally {
            lock.unlock();
         }
      System.out.println();
   }

   static int getCount(){
      return count;
   }
}
```

```
 MyThread thread1 = new
MyThread(obj);
   MyThread thread2 = new
MyThread(obj);
   MyThread thread3 = new
MyThread(obj);
   MyThread thread4 = new
MyThread(obj);
   MyThread thread5 = new
MyThread(obj);
   thread1.start();
   thread2.start();
   thread3.start();
   thread4.start();
   thread5.start();
```

lock.tryLock(),lock.tryLock (timeout, unit)

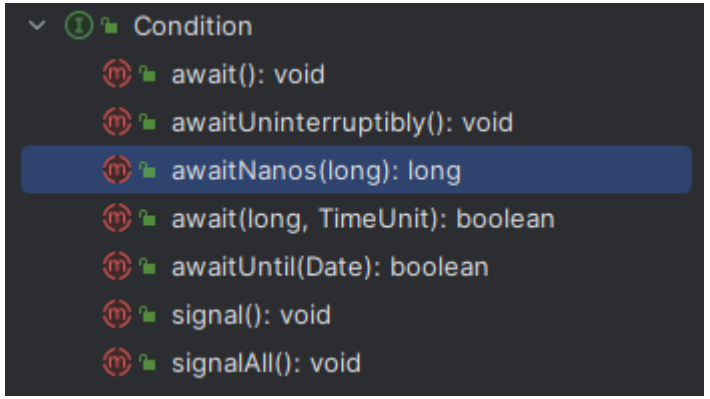
Синхронізовані блоки не мають аналогу tryLock(). Коли потік_2 підходить до критичної секції, доступ до якої заблокований потоком_1, то потік_1 – чекає його задачі припинені.	інтерфейс Lock включає методи: 1. tryLock() – перевіряє можливість отримати доступ до спільного ресурсу і повертає TRUE/FALSE. Коли потік_2 отримує FALSE в результаті виклику tryLock() - потік може виконувати інші задачі. 2. tryLock(timeout, unit)– чекає заданий проміжок часу, після цього, перевіряє можливість отримати доступ до спільного ресурсу

Приклад –які результи в консолі після lock.tryLock(),lock.tryLock (timeout, unit)

```
public class MyThread extends Thread {
   private CommonObject comObjectValue;
   private ReentrantLock relc;
   public MyThread (CommonObject objectValue,
ReentrantLock rl){
      this.comObjectValue=objectValue;
      this.relc=rl;
   }

   public void run() {
boolean lockAcquired = false;
      try {
         lockAcquired = relc.tryLock()||
relc.tryLock(1000, TimeUnit.MICROSECONDS);
         if (lockAcquired) {
            System.out.println(this.getName() + " locked
resource");
            comObjectValue.incCount();
            relc.unlock();
            System.out.println(this.getName() + " releasing
lock(outer lock)");
         }
         else
            System.out.println(this.getName() + "
waiting");
      } catch (InterruptedException e) {
         throw new RuntimeException(e);
      }

   }
      }
```

```
public class CommonObjectLock {
   private static int count =0;

   public  void  incCount() {
      for (int i = 0; i < 10; i++) {
         System.out.print(count++ +
" ");

      }
   }
   static int getCount(){
      return count;
   }
}
```

```
ReentrantLock rel = new
ReentrantLock();
MyThread thread1 = new
MyThread(obj,rel);
MyThread thread2 = new
MyThread(obj,rel);
MyThread thread3 = new
MyThread(obj,rel);
MyThread thread4 = new
MyThread(obj,rel);
MyThread thread5 = new
MyThread(obj,rel);
thread1.start();
thread2.start();
thread3.start();
thread4.start();
thread5.start();
```



Condition newCondition() - повертає об’єкт реалізацію інтерфейса Condition, пов’язаний з об’єктом ReentrantLock. Виклик метода void await() об’єкта Condition заставить поточний потік очікувати до виклику в ньому методів void signal() (void signalAll()).

Таким чином, об'єкт типу «Condition» підтримує ти самі методи (wait(), notify(), notify(All)) які реалізовані в класі Монітор.

При цьому забезпечує додаткові можливості: кілька умов на блокування: один екземпляр блокування може мати кілька пов’язаних екземплярів умови, кожен з яких представляє іншу умову, на яку можуть чекати потоки.

```
boolean aMethod(long timeout, TimeUnit unit)
    throws InterruptedException {
  long nanosRemaining = unit.toNanos(timeout);
  lock.lock();
  try {
    while (!conditionBeingWaitedFor()) {
      if (nanosRemaining <= 0L)
        return false;
      nanosRemaining = theCondition.awaitNanos(nanosRemaining);
    }
    // ...
    return true;
  } finally {
    lock.unlock();
  }
}}</pre>
```

Метод awaitNanos(long) - поточний потік утримує блокування протягом максимального часу, який задано в наносекундах. Реалізація має визначити, чи утримує ще потік блокування на основі умови ***awaitNanos(unit.toNanos(time)) > 0*** і якщо ні, тоді як реагувати - Як правило, створюється виняток (наприклад, IllegalMonitorStateException), і реалізація повинна задокументувати цей факт.

Приклад – які результати в консолі після виклику ReentrantLock із справедливим (fair) або несправедливий (non-fair) доступ?

```
public class Manager {
   private Integer value = null;
   public synchronized int getValue(){
      System.out.println("getValue");
      if (value ==null) {
         try {
           wait(50);

         } catch (InterruptedException e) {
            System.out.println("Thread interrupted while waiting.");
         }
      }
     // System.out.println("Returned value  "+ value);
      return value;
      }
   public synchronized void setValue (int value)
   {
      System.out.println("Inserted value  "+ value);
      this.value = value;
      notify();
   }

   }
```

```
public class ManagerLock {
   private Integer value = null;
   private static ReentrantLock lock = new ReentrantLock();
   private static ReentrantLock lock = new ReentrantLock(true);
   private Condition cond = lock.newCondition();
   public int getValue() {
      System.out.println("getValue");
      try {
         boolean b = lock.tryLock();
         if (b) {
            while (value == null) {
               try {
                  cond.await();
                  System.out.println("waiting");
               } catch (InterruptedException e) {
                  throw new RuntimeException(e);
               }          }        }
         lock.unlock();
      } catch (IllegalMonitorStateException ex) {
         System.out.println(ex.getMessage());
      }

      return value;
   };

   public void setValue(int value) {
      try {
         lock.lock();
        //  System.out.println("Inserted value  "+ value);
         this.value = value;
         cond.signalAll();

   } finally {
         lock.unlock();

      }

   };
}
```

Приклад - condition.awaitNanos(remainingNanos) для операції з матрицями, будемо змінювати час очікування для операцій з матрицями

```
public void setMatrix(int[][] matrix) {
   this.matrix = matrix;
   for (int i=0; i<1000000000; i++) {}
   matrixReady = true;
}


public int[][] getMatrixWithTimeout(long timeoutNanos) throws
InterruptedException {
   lock.lock();
   try {
      long remainingNanos = timeoutNanos;

      while (!matrixReady && remainingNanos > 0) {
         long startTime = System.nanoTime();
         remainingNanos = condition.awaitNanos(remainingNanos);
         long elapsedTime = System.nanoTime() - startTime;
         remainingNanos -= elapsedTime;
      }
      if (!matrixReady) {
         throw new InterruptedException("Timeout occurred while waiting for
matrix");
      }
      return matrix;
   } finally {
      lock.unlock();
   }
}
```

```
private static void testlongmatrixoperation() {
   MatrixOperationExample example = new MatrixOperationExample();
   int[][] matrixData = {
         {1, 2, 3},
         {4, 5, 6},
         {7, 8, 9}
   };
   Thread setThread = new Thread(() -> {
      example.setMatrix(matrixData);
   });
   Thread getThread = new Thread(() -> {
      try {
         waittime=1000000000;
         int[][] retrievedMatrix = example.getMatrixWithTimeout(waittime);
         System.out.println("Retrieved matrix:");
         example.printMatrix(retrievedMatrix);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   });
   setThread.start();
   getThread.start();

   try {
      setThread.join();
      getThread.join();
   } catch (InterruptedException e) {
      e.printStackTrace();
   }
}
```

Interrupt(), IsInterrupted() – як зупинити виконання задачі потоком, якщо викликати Thread.Interrupt()?

Приклад 1. не змінює setMatrix

```
public void setMatrix(int[][] matrix) {
   this.matrix = matrix;
   for (int i=0; i<1000000000; i++) {}
   matrixReady = true;
}
```

Приклад2. змінюємо setMatrix

```
public void setMatrixisInterrupted(int[][] matrix) throws InterruptedException {
   this.matrix = matrix;

   for (int i=0; i<1000000000; i++) {
      if (Thread.currentThread().isInterrupted()){
         System.out.println("stop task");
         matrixReady = false;
         throw new InterruptedException("Timeout occurred while waiting for matrix");

      }
   }
   matrixReady = true;
}
```

```
MatrixOperationExample example = new MatrixOperationExample();
int[][] matrixData = {
      {1, 2, 3},
      {4, 5, 6},
      {7, 8, 9}
};
Thread setThread = new Thread(() -> {
   example.setMatrix(matrixData);
});


Thread getThread = new Thread(() -> {
   try {
      int[][] retrievedMatrix = example.getMatrixWithTimeout(100);
      System.out.println("Retrieved matrix:");
      example.printMatrix(retrievedMatrix);
   } catch (InterruptedException e) {
      e.printStackTrace();
   }
});

setThread.start();
getThread.start();
setThread.interrupt();
try {
   setThread.join();
   getThread.join();
} catch (InterruptedException e) {
   e.printStackTrace();
}
```

Головна особливість ReentrantLock – це блокувати потік більш ніж один раз («повісити замки» Reentrant locking)

```
public class MyReentrantLock {
   public final ReentrantLock lock = new ReentrantLock();
   public void getLock()
   {
      try{
         lock.lock();
         lock.lock();
         System.out.println(Thread.currentThread().getName()+lock.isLocked());
         System.out.println();
         System.out.println(lock.getHoldCount());
}
      finally{
         lock.unlock();
         System.out.println(Thread.currentThread().getName()+ lock.isLocked());
         System.out.println();
         System.out.println(lock.getHoldCount());
         lock.unlock();
         System.out.println(Thread.currentThread().getName()+ lock.isLocked());
         System.out.println();
         System.out.println( lock.getHoldCount());
}
   }
}
```

- ✓ Перший виклик lock.lock(): Потік отримує блокування, якщо воно доступне. Якщо блокування вже утримується іншим потоком, викликаючий потік блокується, доки блокування не стане доступним. Після того, як блокування отримано, воно позначається як утримуване поточним потоком.
- ✓ Подальші виклики lock.lock(): Якщо той самий потік знову викликає lock.lock() для того самого блокування, яке він уже утримує, кількість утримувань блокування збільшується. Блокування залишається утримуваним потоком, і наступні виклики lock.unlock() повинні бути зроблені стільки ж разів, скільки lock.lock(), щоб зняти блокування. Цей механізм дозволяє повторне входження, коли потік може отримати блокування, який він уже утримує, без блокування.
- ✓ Повторне блокування особливо корисне в ситуаціях, коли методам або блокам коду може знадобитися отримати блокування рекурсивно, наприклад, коли метод викликає інший метод, який також потребує блокування. Це допомагає спростити логіку блокування та може покращити читабельність коду та зручність обслуговування. Однак важливо переконатися, що кожен виклик lock.lock() співставляється з відповідним викликом lock.unlock(), щоб уникнути потенційних взаємоблокувань або витоку ресурсів.

ReentrantReadWriteLock - дуже часто використовується в багатопотокових сервісах і кешах, показуючи дуже хороший приріст продуктивності в порівнянні з блоками synchronized. По суті, клас працює в двох взаємовиключних режимах: багато читачів одночасно читають дані і тільки один записувач записує дані.

ReentrantReadWriteLock.ReadLock - ReadLock для читачів, одержуваний через readWriteLock.readLock().

ReentrantReadWriteLock.WriteLock - Write lock для writer'ов, одержуваний через readWriteLock.writeLock().

ReentrantReadWriteLocks можна використовувати для покращення паралельності. Зазвичай це варто лише тоді, коли очікується, що колекції будуть великими, доступ до них матиме більше потоків читачів, ніж потоків записів, і передбачають операції з накладними витратами, які переважають накладні витрати на синхронізацію. Наприклад, ось клас, який використовує TreeMap, який, як очікується, буде великим і до якого доступ буде доступний одночасно.

```
class RWDictionary {
  private final Map<String, Data> m = new TreeMap<>();
  private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
  private final Lock r = rwl.readLock();
  private final Lock w = rwl.writeLock();

  public Data get(String key) {
    r.lock();
    try { return m.get(key); }
    finally { r.unlock(); }
  }
  public List<String> allKeys() {
    r.lock();
    try { return new ArrayList<>(m.keySet()); }
    finally { r.unlock(); }
  }
  public Data put(String key, Data value) {
    w.lock();
    try { return m.put(key, value); }
    finally { w.unlock(); }
  }
  public void clear() {
    w.lock();
    try { m.clear(); }
    finally { w.unlock(); }
  }
}
```

Read Locks: коли потік запитує блокування читання, його можна надати, якщо немає активних блокувань запису. Кілька потоків можуть утримувати блокування читання одночасно, доки немає блокувань запису. Якщо потік запитує блокування читання, але є активне блокування запису, він буде заблоковано, доки блокування запису не буде знято.

Write Locks: коли потік запитує блокування запису, його можна надати, лише якщо немає активних блокувань читання або запису, які утримуються іншими потоками. Це забезпечує ексклюзивний доступ до ресурсу для написання. Якщо є активні блокування читання, утримувані іншими потоками, потік, який запитує блокування запису, буде заблоковано, доки не буде знято всі блокування читання. Крім того, потік, який утримує блокування запису, може отримати блокування читання, не знімаючи блокування запису, таким чином дозволяючи повторний вхід.

Reentrancy: як блокування читання, так і блокування запису підтримують повторне входження, тобто потік, який утримує блокування, може отримати той самий блокування знову, не блокуючи себе. Повторне входження дозволяє вкладене блокування, коли потік може отримати блокування, який він уже утримує, без блокування.

Справедливість: ReentrantReadWriteLock забезпечує як чесні, так і нечесні політики отримання блокувань. У справедливій політиці потоки отримують блокування в тому порядку, в якому вони цього запитали.У несправедливій політиці немає гарантії щодо порядку отримання блокувань, що потенційно може призвести до втрати потоку.

Приклад

```
public class BankAccount {
    private double balance = 0.0;
    private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock readLock = lock.readLock();
    private final Lock writeLock = lock.writeLock();
    public double getBalance() {
        readLock.lock(); // First lock acquisition
        try {
            // Read operation
            return balance;
        } finally {
            readLock.unlock(); // First lock release
        }
    }
    public void deposit(double amount) {
        writeLock.lock(); // Second lock acquisition
        try {
            // Write operation
            balance += amount;
            System.out.println("Deposited: " + amount + ", new balance: " + balance);
        } finally {
            writeLock.unlock(); // Second lock release
        }
    }
    public void withdraw(double amount) {
        writeLock.lock(); // Second lock acquisition
        try {
            // Write operation
            if (balance >= amount) {
                balance -= amount;
                System.out.println("Withdrawn: " + amount + ", new balance: " + balance);
            } else {
                System.out.println("Insufficient funds.");
            }
        } finally {

            writeLock.unlock(); // Second lock release
        }   }}
```

```
BankAccount account = new BankAccount();

// Multiple threads performing operations on the account
Thread thread1 = new Thread(() -> account.deposit(100.0));
Thread thread2 = new Thread(() -> account.withdraw(50.0));
Thread thread3 = new Thread(() -> System.out.println("Balance: " + account.getBalance()));

thread1.start();
thread2.start();
thread3.start();
```

Питання

1. Яке призначення пакета java.util.concurrent.atomic і яку проблему синхронізації він розв’язує?
2. Які основні класи атомарних змінних існують у Java (AtomicInteger, AtomicLong, AtomicBoolean, AtomicReference)?
3. Як працює механізм CAS (Compare-And-Set) і чому він є основою lock-free програмування?
4. У чому полягає різниця між атомарними змінними та використанням synchronized?
5. Яке призначення класу DoubleAccumulator і чим він відрізняється від AtomicDouble (або AtomicLong з приведенням типів)?
6. Як DoubleAccumulator забезпечує кращу масштабованість у системах з великою кількістю потоків?
7. Яку роль відіграє асоціативна функція в роботі DoubleAccumulator?
8. Що таке інтерфейс Future і які можливості він надає для керування асинхронним виконанням задач?
9. Які основні методи інтерфейсу Future (get, cancel, isDone, isCancelled) та їх призначення?
10. Чим ScheduledFuture відрізняється від звичайного Future?
11. Як ScheduledFuture використовується разом із ScheduledExecutorService?
12. Які обмеження інтерфейсу Future усуває клас CompletableFuture?
13. Які методи CompletableFuture використовуються для композиції асинхронних обчислень (thenApply, thenCompose, thenCombine)?
14. Як реалізується обробка винятків у ланцюжках CompletableFuture?
15. У чому полягає відмінність між блокуючим отриманням результату (get, join) та неблокуючою обробкою результатів?