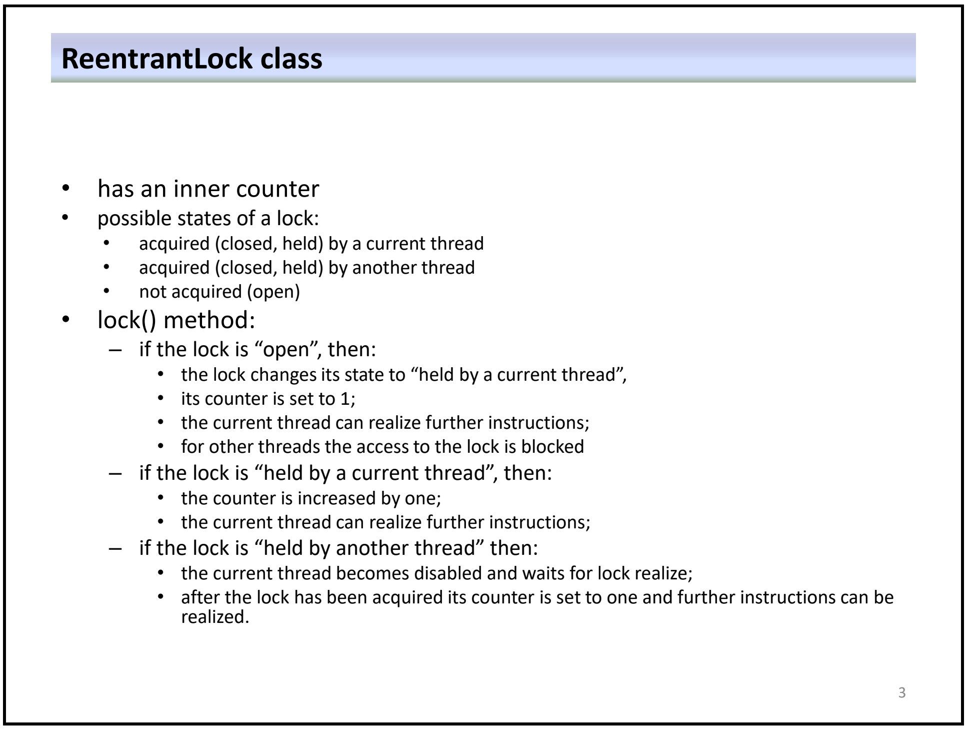
High Level Solutions

Concurrent and parallel programming

Lecture 6.
Academic year: 2018/19

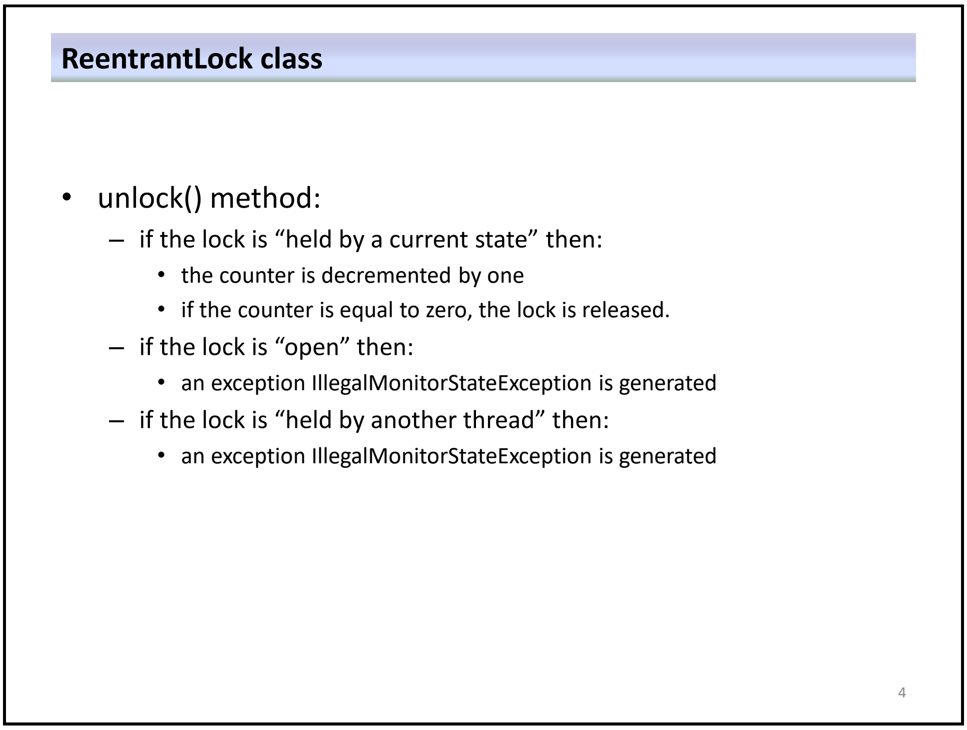
Interface Lock

- Lock – a tool for controlling the access to shared resources
- Defined as an interface in java.util.concurrent.locks
- Two main methods:
  - lock() – acquires the lock
  - unlock() – releases the lock
- Implemented by ReentrantLock class (java.util.concurrent.locks)



ReentrantLock class

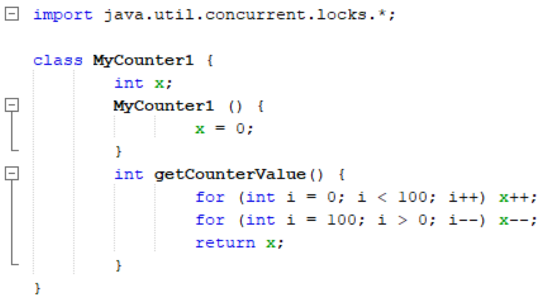
- has an inner counter
- possible states of a lock:
  - acquired (closed, held) by a current thread
  - acquired (closed, held) by another thread
  - not acquired (open)
- lock() method:
  - if the lock is “open”, then:
    - the lock changes its state to “held by a current thread”,
    - its counter is set to 1;
    - the current thread can realize further instructions;
    - for other threads the access to the lock is blocked
  - if the lock is “held by a current thread”, then:
    - the counter is increased by one;
    - the current thread can realize further instructions;
  - if the lock is “held by another thread” then:
    - the current thread becomes disabled and waits for lock realize;
    - after the lock has been acquired its counter is set to one and further instructions can be realized.



ReentrantLock class

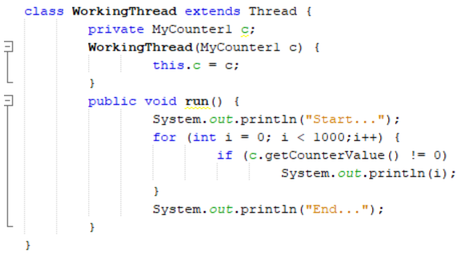
- unlock() method:
  - if the lock is “held by a current state” then:
    - the counter is decremented by one
    - if the counter is equal to zero, the lock is released.
  - if the lock is “open” then:
    - an exception IllegalMonitorStateException is generated
  - if the lock is “held by another thread” then:
    - an exception IllegalMonitorStateException is generated

Example 1 – without synchronization



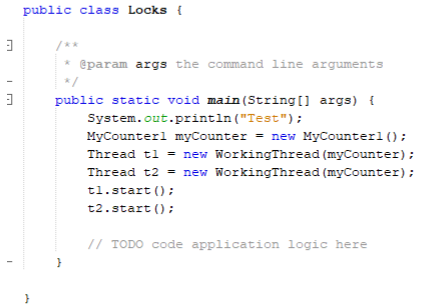
```
import java.util.concurrent.locks.*;

class MyCounter1 {
        int x;
        MyCounter1 () {
                x = 0;
        }
        int getCounterValue() {
                for (int i = 0; i < 100; i++) x++;
                for (int i = 100; i > 0; i--) x--;
                return x;
        }
}
```

Example 1 – without synchronization



```
class WorkingThread extends Thread {
        private MyCounter1 c;
        WorkingThread(MyCounter1 c) {
                this.c = c;
        }
        public void run() {
                System.out.println("Start...");
                for (int i = 0; i < 1000;i++) {
                        if (c.getCounterValue() != 0)
                                System.out.println(i);
                }
                System.out.println("End...");
        }
}
```

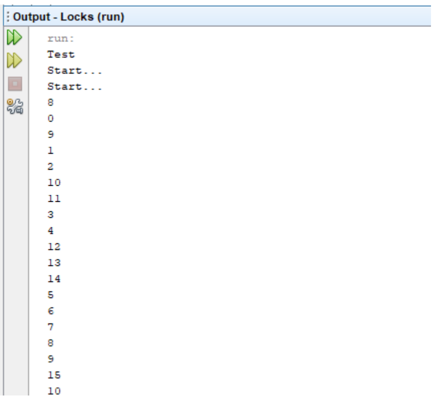
Example 1 – without synchronization



```
public class Locks {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Test");
        MyCounter1 myCounter = new MyCounter1();
        Thread t1 = new WorkingThread(myCounter);
        Thread t2 = new WorkingThread(myCounter);
        t1.start();
        t2.start();

        // TODO code application logic here
    }

}
```

7

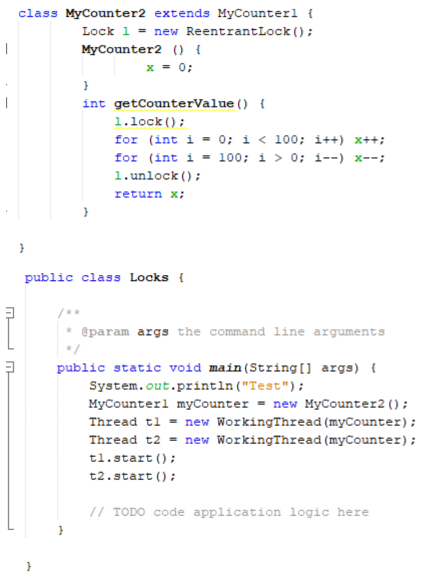
Example 1 – without synchronization



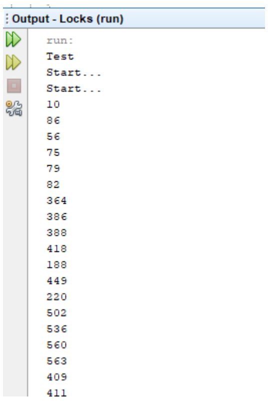
```
run:
Test
Start...
Start...
8
0
9
1
2
10
11
3
4
12
13
14
5
6
7
8
9
15
10
```

8

Example 2 – with improper synchronization



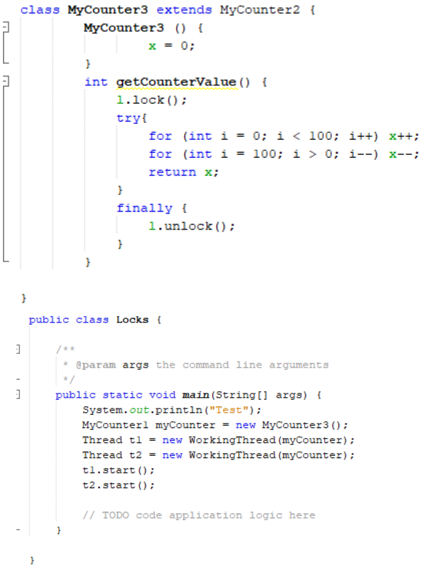
```
class MyCounter2 extends MyCounter1 {
        Lock l = new ReentrantLock();
        MyCounter2 () {
                x = 0;
        }
        int getCounterValue() {
            l.lock();
            for (int i = 0; i < 100; i++) x++;
            for (int i = 100; i > 0; i--) x--;
            l.unlock();
            return x;
        }

}

 public class Locks {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Test");
        MyCounter1 myCounter = new MyCounter2();
        Thread t1 = new WorkingThread(myCounter);
        Thread t2 = new WorkingThread(myCounter);
        t1.start();
        t2.start();

        // TODO code application logic here
    }

 }
```



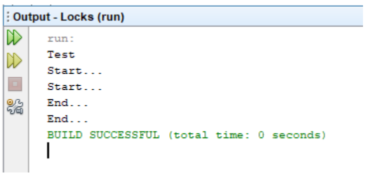
Output - Locks (run)

```
run:
Test
Start...
Start...
10
86
56
75
79
82
364
386
388
418
188
449
220
502
536
560
563
409
411
```

Example 3 – with proper synchronization



```
class MyCounter3 extends MyCounter2 {
        MyCounter3 () {
                x = 0;
        }
        int getCounterValue() {
            l.lock();
            try{
                for (int i = 0; i < 100; i++) x++;
                for (int i = 100; i > 0; i--) x--;
                return x;
            }
            finally {
                l.unlock();
            }
        }

}
 public class Locks {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Test");
        MyCounter1 myCounter = new MyCounter3();
        Thread t1 = new WorkingThread(myCounter);
        Thread t2 = new WorkingThread(myCounter);
        t1.start();
        t2.start();

        // TODO code application logic here
    }

 }
```



Output - Locks (run)

```
run:
Test
Start...
Start...
End...
End...
BUILD SUCCESSFUL (total time: 0 seconds)
```

tryLock() method

- public boolean **tryLock**()
  if the lock is “open”, then:
  - the lock changes its state to “held by a current thread”,
  - its counter is set to 1;
  - the current thread can realize further instructions;
  - for other threads the access to the lock is blocked
- if the lock is “held by a current thread”, then:
  - the counter is increased by one;
  - the current thread can realize further instructions;
- if the lock is “held by another thread” then:
  - method will return immediately with the value **false**
  - the current thread can realize further instructions.

tryLock() method

- public boolean **tryLock**(long timeout, TimeUnit unit) throws InterruptedException

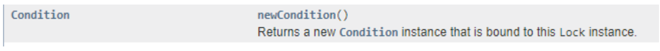
- if the lock is “held by another thread” then current thread becomes disabled for thread scheduling purposes and lies dormant until one of three things happens:
  - the lock is acquired be the current thread,
  - Some other thread interrupts the current thread; or
  - The specified waiting time elapses.

Conditions

- Condition – a mechanism allowing to suspend and resume an execution of a critical section in specified circumstances (similar to wait/notify/notifyAll)

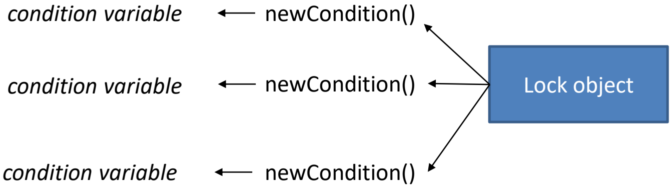
Conditions

In Lock interface:



Condition	newCondition() Returns a new Condition instance that is bound to this Lock instance.

returns a Condition object associated with a given Lock object.



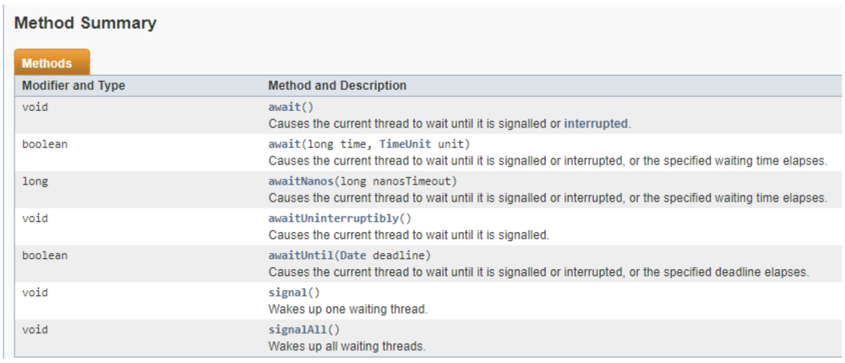
condition variable ← newCondition()

condition variable ← newCondition()

condition variable ← newCondition()

Lock object

Condition interface

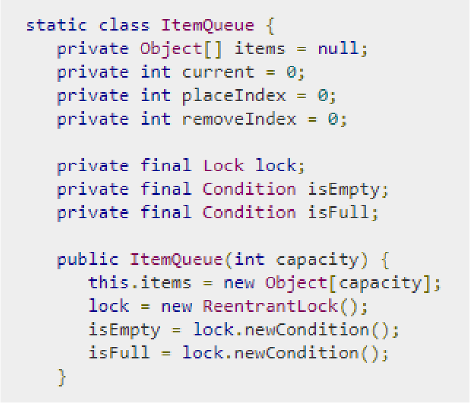


Method Summary

Methods

Modifier and Type	Method and Description
void	await() Causes the current thread to wait until it is signalled or interrupted.
boolean	await(long time, TimeUnit unit) Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.
long	awaitNanos(long nanosTimeout) Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.
void	awaitUninterruptibly() Causes the current thread to wait until it is signalled.
boolean	awaitUntil(Date deadline) Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses.
void	signal() Wakes up one waiting thread.
void	signalAll() Wakes up all waiting threads.

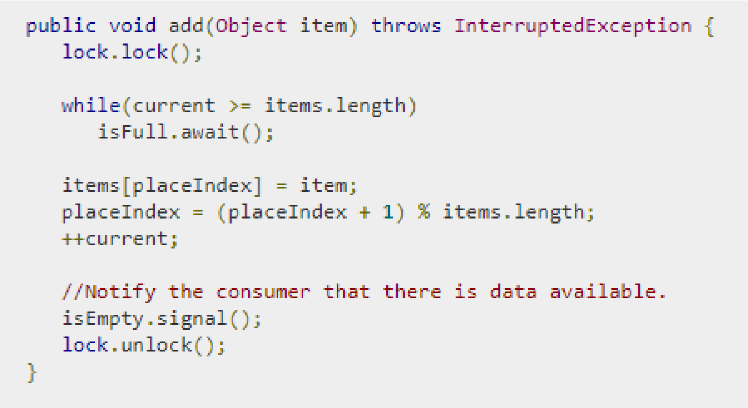
Condition usage – Consumer-Producer problem



```
static class ItemQueue {
   private Object[] items = null;
   private int current = 0;
   private int placeIndex = 0;
   private int removeIndex = 0;

   private final Lock lock;
   private final Condition isEmpty;
   private final Condition isFull;

   public ItemQueue(int capacity) {
      this.items = new Object[capacity];
      lock = new ReentrantLock();
      isEmpty = lock.newCondition();
      isFull = lock.newCondition();
   }
```

Source: https://www.tutorialspoint.com/java_concurrency/concurrency_condition.htm

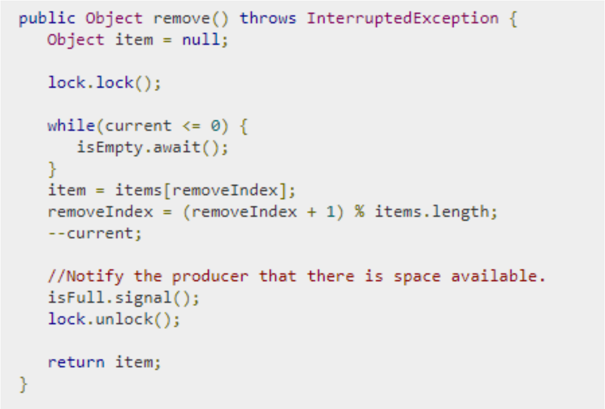
Condition usage – Consumer-Producer problem



```
public void add(Object item) throws InterruptedException {
   lock.lock();

   while(current >= items.length)
      isFull.await();

   items[placeIndex] = item;
   placeIndex = (placeIndex + 1) % items.length;
   ++current;

   //Notify the consumer that there is data available.
   isEmpty.signal();
   lock.unlock();
}
```

Source: https://www.tutorialspoint.com/java_concurrency/concurrency_condition.htm

Condition usage – Consumer-Producer problem



```
public Object remove() throws InterruptedException {
   Object item = null;

   lock.lock();

   while(current <= 0) {
      isEmpty.await();
   }
   item = items[removeIndex];
   removeIndex = (removeIndex + 1) % items.length;
   --current;

   //Notify the producer that there is space available.
   isFull.signal();
   lock.unlock();

   return item;
}
```

Source: https://www.tutorialspoint.com/java_concurrency/concurrency_condition.htm