Synchronization, Locks, Conditional Variables in Java

Prince Jha

·(https://medium.com/@jhaprincet/synchronization-locks-conditional-variables-in-java-b0c3413fd28e)

Follow

18 min read
·
Dec 13, 2022

[[Engineer Java Concurrency — 3]]

- **Race Condition**
- **Lock objects**
- **Condition Variables** — *await(), signal(), signalAll()*
- **synchronized keyword**
- **synchronized blocks**
- **The intrinsic condition variable** — *wait(), notify(), notifyAll()*
- **synchronized in static context**

Race Condition:

When multiple threads share same data and try to modify the data, it is possible that the modifications could lead to an inconsistent state or corrupted state of the object that depends on the order in which the threads operate on the object, such a situation is called **Race Condition**.

Example of Race Condition:

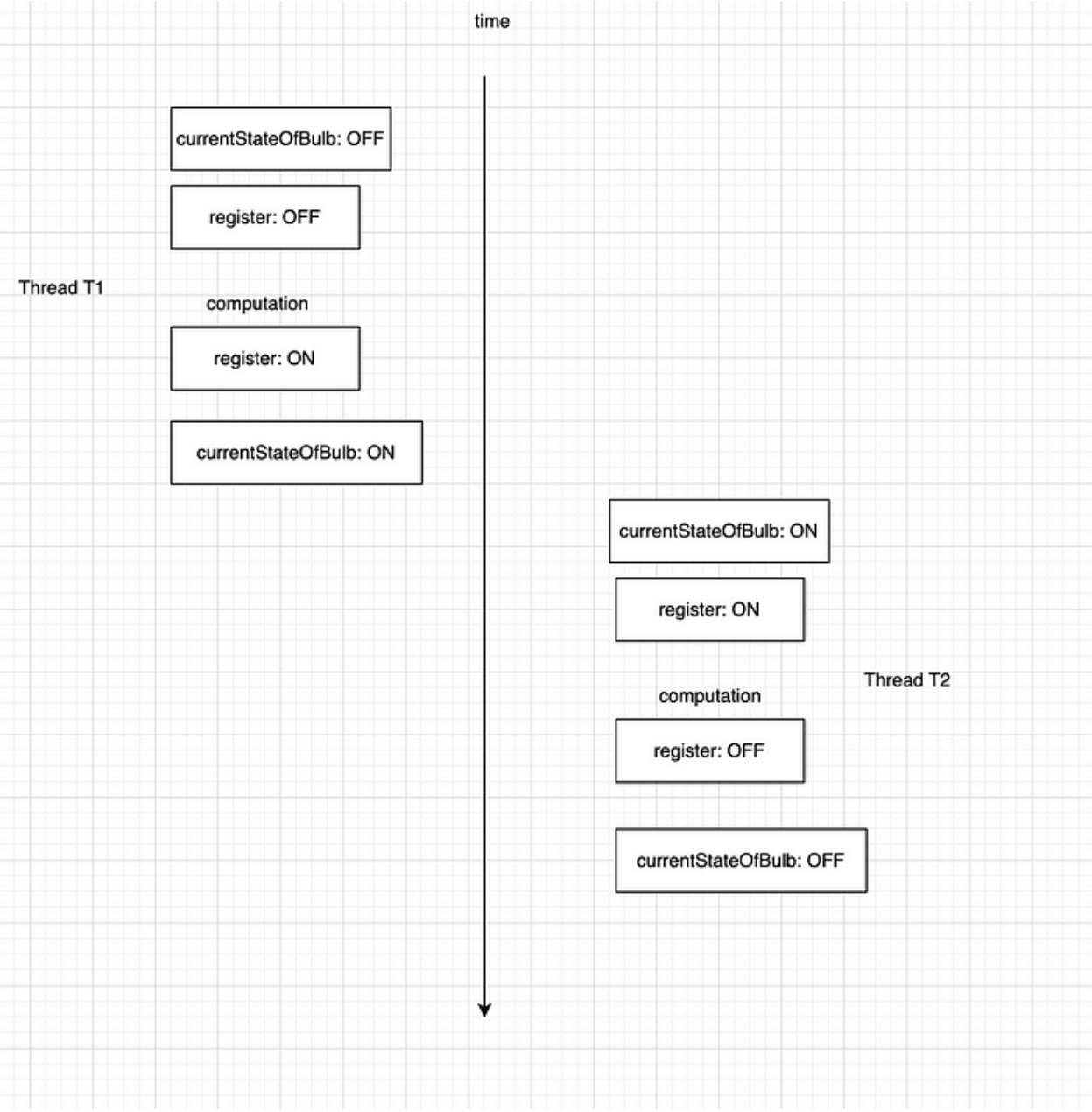
Let's say we have a single bulb and there are two threads try to change its state at the same time. So if the bulb was initially OFF, you might expect that overall switching the bulb twice will bring it back to the OFF state.

```
class Bulb {
   private static enum STATE {
       OFF,
       ON;
   }

   private STATE currentStateOfBulb = STATE.OFF;

   public void toggleState() {
       STATE before = currentStateOfBulb; // ....(*)
       STATE result = STATE.OFF;
       System.out.println("WHAT WILL HAPPEN HERE");
       if(before == STATE.OFF) {
           result = STATE.ON;
       }

       currentStateOfBulb = result; // ....(#)
       printState();
   }

   private void printState() {
        System.out.println("switch press by thread " +
           Thread.currentThread().getName() +
           " resulting state " + currentStateOfBulb
       );
   }
}

public class MyClass {

    public static void main(String [] args) {
        Bulb bathroomBulb = new Bulb();
        Thread thread1 = new Thread(() -> {
          bathroomBulb.toggleState();
        });
        thread1.setName("PRINCE THREAD");

        Thread thread2 = new Thread(() -> {
            bathroomBulb.toggleState();
        });
        thread2.setName("TUSHAR THREAD");

        thread1.start();
        thread2.start();
    }

}
```

```
WHAT WILL HAPPEN HERE
WHAT WILL HAPPEN HERE
switch press by thread PRINCE THREAD resulting state ON
switch press by thread TUSHAR THREAD resulting state ON
```

The output above ended up keeping the bulb ON even after two threads each tried switching it once.

Expected Behaviour: One of the thread switches the bulb from OFF to ON, the other thread switches the bulb from ON to OFF.



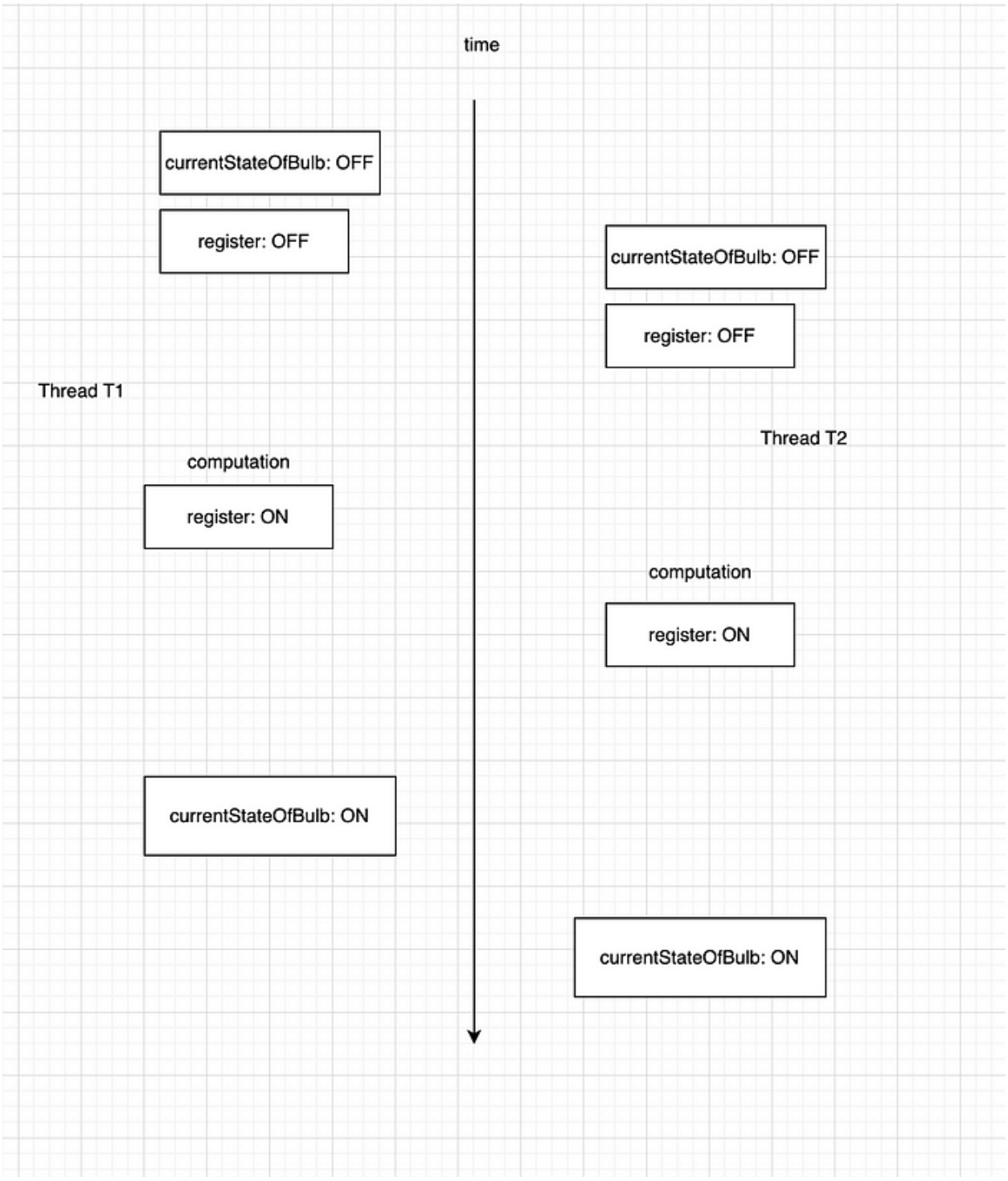
Expected Behaviour

Actual Behaviour: One possible scenario that might occur,
The first thread reads the state of the object to be OFF state at line (*), the first thread has not yet executed the line (#) and hence the state of the bulb is OFF still.

The second thread meanwhile reads the state of the object to be OFF (since the object property still holds the value OFF).

The two threads both end up writing the value ON to the object property. Hence even though two switches occur, result appears to be equivalent to one switch.

Actual Behaviour:



Actual Behaviour

Synchronization: Multiple threads should access shared data in a way that leaves the object state in a consistent state.

What went wrong in the program?
The second thread executed the line (*) before the first thread executes the line (#). i.e., the second thread reads the state of the object before the first thread has written it.

To Fix the program we have to ensure that threads can exclusively access the object. ie., we need to ensure that the thread T2 should not be allowed to operate on the object until thread T1 has finished operating on the object.

Lock object:

A lock object can be used to synchronize access to executing certain lines of code. Only one thread that has obtained the lock object will be allowed to execute certain lines of code whereas the other threads that try to execute lines of code but that failed to acquire the lock object will get blocked.

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

class Bulb {
   private ReentrantLock lockObj = new ReentrantLock();
   private static enum STATE {
       OFF,
       ON;
   }

   private STATE currentStateOfBulb = STATE.OFF;

   public void toggleState() {

       lockObj.lock(); // .......(***)
       try {
          STATE before = currentStateOfBulb;
          STATE result = STATE.OFF;
          System.out.println("WHAT WILL HAPPEN HERE");
          if(before == STATE.OFF) {
             result = STATE.ON;
          }

          currentStateOfBulb = result;
          printState();
       } finally {
           lockObj.unlock(); // .....(****)
       }
   }

   private void printState() {
        System.out.println("switch press by thread " +
           Thread.currentThread().getName() +
           " resulting state " + currentStateOfBulb
       );
   }
}
```

```
public class MyClass {

    public static void main(String [] args) {
        Bulb bathroomBulb = new Bulb();
        Thread thread1 = new Thread(() -> {
          bathroomBulb.toggleState();
        });
        thread1.setName("PRINCE THREAD");

        Thread thread2 = new Thread(() -> {
            bathroomBulb.toggleState();
        });
        thread2.setName("TUSHAR THREAD");

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

We have synchronized access to executing lines of code that by first locking on the Lock object. Thread T1 acquires the lock and hence can continue executing those lines of the code that appear after the *lock()* method call. On the other hand when Thread T2 tries to acquire the lock it will be blocked since the lock is currently acquired by T1.
T1 has the lock and continues the execution. After updating the bulb state from OFF to ON, it finally releases the Lock by calling the *unlock()* method.
The thread scheduler allows thread T2 to acquire the lock and hence the thread becomes Runnable again. Once the thread T2 get scheduled for execution it continues the execution of the lines following the *lock()* method call. T2 now has the lock with it. T2 reads the state of the bulb to be ON. The thread T2 updates the bulb state from ON to OFF. The thread T2 releases its lock by calling the *unlock()* method.

The section of the code whose execution is protected by using a lock object is called critical section. There could be multiple critical sections protected by using the same lock. Only one thread will be allowed inside the critical section. Other threads will be blocked till they get to acquire the lock.

A few points to understand:

- Not every thread that calls the lock() will be granted access to the Lock object immediately.
- When there are multiple threads waiting to acquire a Lock object in the Blocked state, the thread that will acquire the Lock object once the Lock object is available is decided by the thread scheduler.
- At the most one thread can own a Lock object at a point in time.
- The same Lock object should be used to synchronize those critical sections of code that can change the state of an object.
- It is common to use one Lock to synchronize code written in multiple methods when they operate on the same object.

Reentrant Locks:

A thread which already has acquired the Lock object by calling the *lock()* can keep calling the *lock()* method to re-acquire the lock. In simple words, there is a counter associated with the Lock object which counts the number of times the Lock is acquired by the thread that has it. An *unlock()* should be called corresponding to each *lock()* call.

When the count corresponding to the Lock object reaches zero, the Lock is said to be released and hence is available to other threads. One of these lucky threads decided by the scheduler will get a chance to execute the critical section.

IMPORTANT NOTE:

1. Always call the *unlock()* corresponding to a *lock()* on the Lock object otherwise the lock count will remain greater than one and thus all the other threads that are blocked will never get a chance to execute the critical section and get blocked forever.
2. Sometimes a thread could terminate due to an exception inside the critical section, it is essential to write a try-finally block around the critical section and ensure that the critical section is inside a try block, and the lock is released in the finally block so that the lock is released by the thread T and made available to the other blocked threads.

```
lockObj.lock(); // obtain the Lock lockObj, increment lock counter
try {
  // critical section
} finally {
    // irrespective of exception or no exception
    lockObj.unlock(); // decrement lock counter, release lock if counter == 0
}
```

Re-writing the bulb program to demonstrate the reentrant lock:

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

class Bulb {
   private ReentrantLock lockObj = new ReentrantLock();
   private static enum STATE {
       OFF,
       ON;
   }

   private STATE currentStateOfBulb = STATE.OFF;

   public void toggleState() {

       lockObj.lock(); // .......(***)
       try {
          STATE before = currentStateOfBulb;
          STATE result = STATE.OFF;
          System.out.println("WHAT WILL HAPPEN HERE");
          if(before == STATE.OFF) {
```

```
            result = STATE.ON;
          }

          currentStateOfBulb = result;
          printState();
        } finally {
            lockObj.unlock(); // .....(****)
        }
    }

    private void printState() {
         lockObj.lock(); // .......(***)
         try {
            System.out.println("switch press by thread " +
            Thread.currentThread().getName() +
            " resulting state " + currentStateOfBulb
            );
         } finally {
              lockObj.unlock(); // .....(****)
         }

    }
}

public class MyClass {

     public static void main(String [] args) {
         Bulb bathroomBulb = new Bulb();
         Thread thread1 = new Thread(() -> {
           bathroomBulb.toggleState();
         });
         thread1.setName("PRINCE THREAD");

         Thread thread2 = new Thread(() -> {
             bathroomBulb.toggleState();
         });
         thread2.setName("TUSHAR THREAD");

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

In the above program the thread T1 that obtain the Lock lockObj first in the toggleState method will again acquire the same lock in the printState method. The thread T1 at this point in time has acquired the lock two times. The critical section of printState prints the bulb state, after the critical section of printState, the counter is decremented from two back to one when *unlock()* is called. The lock is still with thread T1. The thread T1 completes execution of critical section of toggleState method. The thread T1 releases the Lock lockObj when *unlock()* is called since the counter decrements from one to zero.

The thread T2 which is still blocked and trying to acquire the lock in toggleState method will now obtain the available Lock lockObj and so on.

To the readers who are curious to know why I did not lock twice in my earlier code, the reason is that the method Bulb::printState is only called from inside the critical section of Bulb::toggleState which is already synchronized and hence the method Bulb::toggleState will only be executed by one thread which holds the lock.

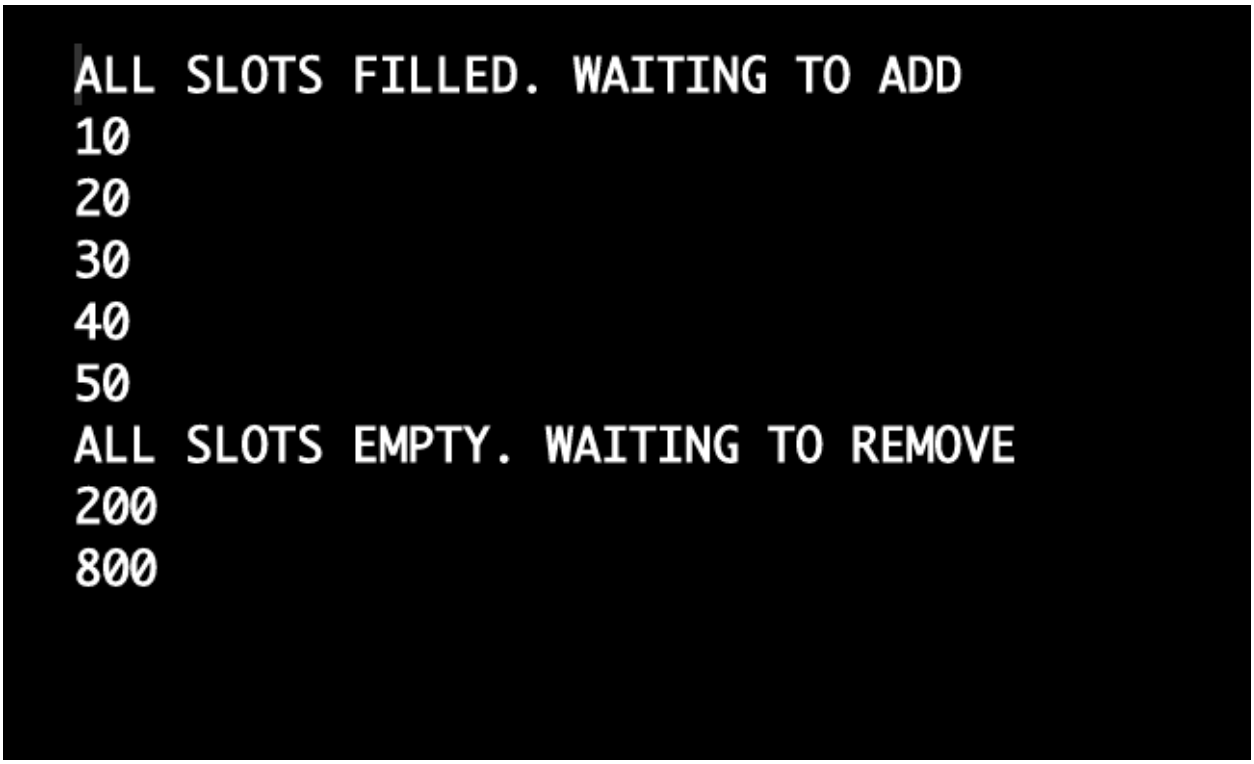
Condition Variables:

Often a thread wants to perform certain operations only when a predicate/ condition holds true. The thread cannot continue to perform meaningful computation until the condition becomes true. We can use condition variables in such cases.

Example: Design a queue of fixed size N. The add operation on a queue needs to wait until there is some empty space in the queue. On the other hand the remove operation should wait until there is at least one element at the head of the queue.

```
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;
import java.util.stream.*;
import java.util.*;

class MyQueue<T> {
   private static int EMPTY_QUEUE_SIZE = 0;

   private Queue<T> queue = new ArrayDeque<>();
   private int maxSize;
   private ReentrantLock lockObj = new ReentrantLock();
   private Condition queueContainsSpace = lockObj.newCondition();
   private Condition queueContainsElement = lockObj.newCondition();
   public MyQueue(int n) {
      maxSize = n;
   }

   public void add(T element) {
       lockObj.lock(); // lock the associate Lock
       try {
           while(queue.size() == maxSize) {
              System.out.println("ALL SLOTS FILLED. WAITING TO ADD");
              queueContainsSpace.await();
              // wait for notification
              // loop ensures that the queue has empty slots when loop
terminates
           }
           queue.add(element);
           queueContainsElement.signalAll();
           // signal when the queue state changes
           // threads waiting for notification will be woken and
           // made runnable and get blocked ...

       } catch(InterruptedException excetion) {
          Thread.currentThread().interrupt(); // set interrupted flag
        } finally {
           lockObj.unlock(); // unlock after critical section is complete
```

```
        }
    }

    public T remove() {
         T result = null;
         lockObj.lock();
         try {
           while(queue.size() == EMPTY_QUEUE_SIZE) {
             System.out.println("ALL SLOTS EMPTY. WAITING TO REMOVE");
             queueContainsElement.await();
           }
           result = queue.element();
           queue.remove();
           queueContainsSpace.signalAll();
         } catch(InterruptedException excetion) {
           Thread.currentThread().interrupt(); // set interrupted flag
         } finally {
            lockObj.unlock();
         }
         return result;
    }
}

public class MyClass {
    public static void main(String [] args) {
         MyQueue<Integer> myQueue = new MyQueue<>(5);
         Thread threadA = new Thread(() -> {

             Stream.
             of(10, 20, 30, 40, 50).
             forEach(myQueue::add);

             myQueue.add(200);
             myQueue.add(800);
         });


         Thread threadB = new Thread(() -> {
             Stream.
             iterate(0, i -> i < 7, i -> i + 1).
             forEach((i) -> {
                 Integer value = myQueue.remove();
                 System.out.println(value);
             });

         });

         threadA.start();
         threadB.start();

    }

}
```



The design of the custom queue is such that a queue has a Lock object and the Lock has two condition variables associated with it.
The condition when the queue size is not full and hence can accommodate a new entry is conceptually/logically represented using the queueContainsSpace condition variable.
The condition when the queue is not empty and hence an element is available in the queue for removal is conceptually/logically represented using the queueContainsElement condition variable.

Imagine the situation when a thread tries to add an element to the shared queue and the queue is full, lets assume that the thread T1 obtains a lock and starts executing the critical section of add method, the while loop condition is true since the queue is full and hence the *await()* method is called on the condition variable. As per the problem statement the thread T1 should not continue its execution since all the slots in the queue are filled.
But if the thread T1 keeps waiting for the condition to get better but at the same time holds the lock then the situations can never get better.
For situations to become better the thread T1 needs to release the lock so that some other thread can gain the lock and operate on the queue. The thread T1 should hope that the other thread that gets a chance to operate on the queue makes situations better for it.

The *await()* method call on a condition variable makes the thread T1 go to a wait set and changes the thread state from Runnable to Waiting. The thread T1 releases the lock before going to the wait set. The thread T1 does it because the thread T1 wants to ensure that its execution should only continue once the condition is true. The other threads will now compete for the available lock and one of the thread will obtain the lock and continue executing its critical section. The thread T1 has put all its faith in the other threads and is waiting in the wait set. Once the other

thread brings the object into a state that could be beneficial to remaining threads it should *notifyAll* the waiting threads. The only way that thread T1 can come out of the wait set is when it receives notification on the condition variable which led to its waiting state.
Once the thread T1 receives notification. It is made Runnable and then gets blocked when it tries to acquire the Lock. Whenever the lock is available and it obtains the lock it continues the execution of the next instruction from the instruction due to which it went into the wait set.

A Lock object could have multiple condition variables associated with it. The general form of using the condition variable is

```
lockObj.lock(); // increment lock count, critical section ahead
try {

    while( codition is not ok to proceed) {
        conditionVariable.await();
        // release lock
        // go to wait set
        // wait for notification from other thread hoping codition will be
better
    }

    // logic of critical section

} catch(InterruptedException error) {
    // exception when interrupted inactive
} finally {
   lockObj.unlock(); //decrement lock counter, release if count == 0
}
```

Continuing our assumed example, the thread T1 cannot proceed to add elements since queue is full and hence it waits until the queue has some space. Note that the only way the queue can have space at this point in time by allowing the removal of elements. So the thread T1 releases its lock and goes to wait set. It waits for notification on the queueContainsSpace condition variable.

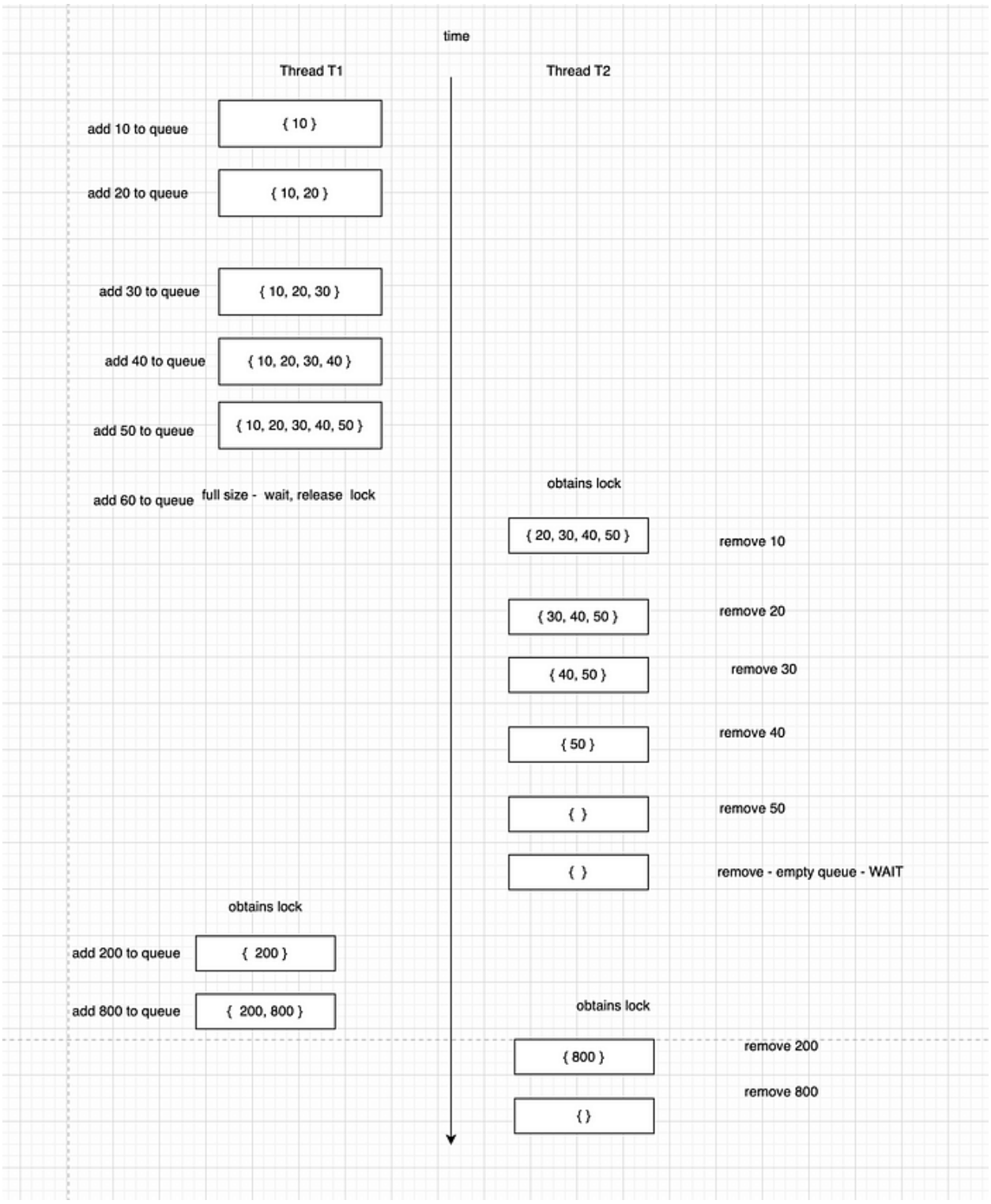
The released lock can be obtained by another thread that could remove an element from the queue, when the remove method executes, *queueContainsSpace.signalAll();* line will actually cause all the threads waiting in the wait set due to queueContainsSpace condition variable to be removed from the wait set and will make them Runnable and then Blocked. The thread that sends notification will continue to keep the lock. These blocked threads try to compete for Lock object since these threads need to execute the critical section. When the Lock is available one of them gets the lock and continues execution whereas others that do not get the lock will remain in the Blocked state.

There is also a *signal* method that can be called on a condition variable. The signal method will randomly pick exactly one thread out of all the threads that are waiting in the wait set on the condition variable. If the signalled thread still cannot proceed it will release the lock and wait in the wait-set. If the thread that obtains the lock does not send notification on the same condition variable, then all the threads waiting in the wait-set will never get a chance of execute again.

Again continuing the assumed example, the thread T1 waits for some other thread to make space in the queue by removing an element. It waits for the notification which it receives when *signalAll* is called. It can then continue to add an element since now there is space in the queue.

Output Explanation:

threadA adds five elements to the queue. The queue gets full. Adding the sixth element is not allowed since max size is five, hence the threadA releases the lock and waits in the wait set for notification till queueContainsSpace. The threadB starts removing elements from the queue. Whenever threadB removes elements from queue the queueContainsSpace and hence the notification is sent to all threads waiting on that condition variable. The threadA receives the notification and hence it is made Runnable and gets blocked since it tries to acquire the lock. The threadB continues its execution and removes all the 5 elements. When threadB tries removing the 6th element, the queue is empty so now threadB releases the lock and waits in the wait set until queueContainsElement. threadA obtains the lock and continues the execution and adds two more elements. When threadA adds elements to the queue the queueContainsElement and hence notification is sent to all the threads that are waiting on that condition variable. threadB receives the notification and is made Runnable and is blocked when it tries to acquire the lock. threadA completes adding two elements and releases the lock. threadB acquires the lock and removes two elements.



synchronized keyword in Java:

The methods of a class can be marked with the synchronized keyword indicating that the method body is a critical section.
Every Java object has an associated Lock object and that Lock object has an associated condition variable.

```
class A {
   public synchronized void method() {
       // code in critical section...
   }
}
```

The above code is same as

```
class A {
   private ReentrantLock intrinsicLockObj = new ReentrantLock();
   public void method() {
       intrinsicLockObj.lock(); // only enter the critical section once you
own a lock
       try {
          // code in the critical section
       } finally {
          intrinsicLockObj.unlock();
       }
   }
}
```

In other words, all the methods that are synchronized can only be executed by a thread once the intrinsic lock of the object is acquired by that thread.

If there are several methods in the class marked with the synchronized keyword, then only that one thread which gains the lock can execute these methods by acquiring the intrinsic reentrant lock, all the other threads are blocked until they get the lock and get a chance to execute.

```
class Address {

  private String pinCode = "";
  private String streetName = "";
  private String areaName = "";

  public synchronized void changeAddress(String pin, String street, String
area) {
      changePinCode(pin);
      changeStreetName(street);
      changeAreaName(area);
  }

  private synchronized void changePinCode(String otherPin) {
      pinCode = otherPin;
  }

  private synchronized void changeStreetName(String otherStreet) {
      streetName = otherStreet;
  }
```

```
  private synchronized void changeAreaName(String otherArea) {
       areaName = otherArea;
  }
}
```

Can be considered to be same as

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

class Address {
  private ReentrantLock intrinsicLockObj = new ReentrantLock();
  private String pinCode = "";
  private String streetName = "";
  private String areaName = "";

  public void changeAddress(String pin, String street, String area) {
      intrinsicLockObj.lock();
      try {
          changePinCode(pin);
          changeStreetName(street);
          changeAreaName(area);
      } finally {
          intrinsicLockObj.unlock();
      }
  }

  private void changePinCode(String otherPin) {
      intrinsicLockObj.lock();
      try {
         pinCode = otherPin;
      } finally {
         intrinsicLockObj.unlock();
      }
  }

  private void changeStreetName(String otherStreet) {
      intrinsicLockObj.lock();
      try {
           streetName = otherStreet;
      } finally {
          intrinsicLockObj.unlock();
      }
  }

  private void changeAreaName(String otherArea) {
       intrinsicLockObj.lock();
       try {
          areaName = otherArea;
       } finally {
           intrinsicLockObj.unlock();
       }
  }

}
```

synchronized block:

Using synchronized keyword to protect critical section by locking on a Java object. A block of statements can be constituted to be a critical section of code and can be protected by Locking on a Java object. Internally it means that the reentrant lock associated with the Java object will be used to synchronize execution of lines of code. The block of statements is called a synchronized block.

```
class A {
     private int x;
     private int y;

     public void changeX(int otherVal) {
         System.out.println("Outside Critical Section");
         synchronized(this) {
             System.out.println("Other val " + otherVal);
             this.x = otherVal;
         }

         System.out.println("Outside the synchronized block");
     }
}
```

The above synchronized block is equivalent to the following

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

class A {
     private int x;
     private int y;
     private ReentrantLock intrinsicLockObj = new ReentrantLock();

     public void changeX(int otherVal) {
         System.out.println("Outside Critical Section");
         this.intrinsicLockObj.lock(); // lock the intrinsic associated lock
         try {
             System.out.println("Other val " + otherVal);
             this.x = otherVal;
         } finally {
              this.intrinsicLockObj.unlock();
         }

         System.out.println("Outside the synchronized block");
     }
}
```

The associate condition variable:

The intrinsic condition variable associated to the intrinsic lock associated to a Java object can be used by a thread to *wait* on a notification, *notify* one of the waiting threads on that condition

variable, *notifyAll* the waiting threads on the intrinsic condition variable.
These methods are equivalent to the *await(), signal(), signalAll()* methods that we have seen earlier with the condition variable.

```
class Bank {
     private int [] accountBalance;

    Bank(int numberOfAccounts) {
        accountBalance = new int [numberOfAccounts];
    }

    public synchronized void transferAmount(int fromAccountNumber, int
toAccountNumber, int amount) {
        try {
            while(accountBalance[fromAccountNumber] < amount) {
               wait();
              // release the lock
              // wait in a wait set for notification
           }
           accountBalance[toAccountNumber] += amount;
           accountBalance[fromAccountNumber] -= amount;
           notifyAll();
          // since account balance changed, situations could get better for
threads
          // notify threads that are waiting for notification
        } catch(InterruptedException exception) {
            Thread.currentThread().interrupt(); // set interrupt flag
        }

    }
}
```

The above code is similar to

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

class Bank {
     private ReentrantLock intrinsicLockObj = new ReentrantLock();
     private Condition intrinsicCondition = intrinsicLockObj.newCondition();
     private int [] accountBalance;

    Bank(int numberOfAccounts) {
        accountBalance = new int [numberOfAccounts];
    }

    public void transferAmount(int fromAccountNumber, int toAccountNumber,
int amount) {
           intrinsicLockObj.lock(); // obtain the lock
           try {
                while(accountBalance[fromAccountNumber] < amount) {
                    intrinsicCondition.await();
                    // release the lock
                    // wait in a wait set for notification from other thread
                }
```

```
                accountBalance[toAccountNumber] += amount;
                accountBalance[fromAccountNumber] -= amount;
                intrinsicCondition.signalAll();
                // since account balance changed, situations could get better
for threads
                // notify threads that are waiting for notification
                // notified threads will be made Runnable and then get
Blocked

            } catch(InterruptedException exception) {
                Thread.currentThread().interrupt(); // set interrupt flag
            } finally {
                intrinsicLockObj.unlock(); // unlock after critical section
completes
            }
    }

}
```

The working of the above code is such that a thread T that wants to transfer amount from one bank account to another bank account should first obtain the lock on the Bank object. (or rather the intrinsic lock associated with the bank object). It should get suspended and wait in a wait set when the source bank account does not have enough funds. It is necessary that the condition variable *wait*() is inside a while statement since we want to ensure that when thread wakes up it again attempts to ensure that the source bank account has the sufficient amount or not. When a thread gets suspended due to insufficient amount, it releases the lock and goes to the waiting state. The thread scheduler can pick one thread from the other threads that are blocked and are competing for the lock to execute a critical section protected by the same lock. Whenever the account balance changes, a notification will be sent to all the threads that are waiting in the wait set using the *notifyAll*() and they will all wake up and try to obtain the lock and they will all get Blocked competing for the lock. Once one of these threads gets the lock (whenever the lock becomes available) it will resume the execution of code exactly from the next instruction from the instruction whose execution caused it to go to the waiting state.

Synchronizing static methods:

We know that the *this* context is not available inside the static methods of a class. So how do we protect critical section of code from being executed simultaneously by multiple threads that is present inside static methods.

The idea is that the static methods of a class are defined on the type and hence it makes sense to use the *Class* object of that type for the sake of locking.
synchronized instance method lock on the intrinsic lock associated with the *this* context object.
synchronized static methods lock on the intrinsic lock associated with the Class<T> object where T is the type on which static method is defined.

```
class A {
    public static synchronized void method() {
         // method definition
```

```
    }
}
```

is same as

```
class A {
    public static void method() {
        final Class<A> classObj = A.class;
        synchronized(classObj) {
             // method definition
        }
    }
}
```

which is same as locking on the intrinsic lock associated with the *classObj*.

Instead of synchronizing the entire method it is also possible to synchronize only a block of statements, such blocks are called synchronized blocks.

```
class X {
    public static void method() {
       System.out.println("Welcome, Non synchronized line");
       System.out.println("Critical section ahead");
       synchronized(X.class) {
           // lines of code in critical section, thread should acquire lock
       }

    }
}
```

Example of synchronized static block:

```
class Singleton {
   private volatile static Singleton instance = null;
   // private constructor to avoid calling new from outside the class

   // declare the fields that you need in your Signleton/subclass of
Singleton

   private Singleton() {
     // initialize the fields of the object
   }

   public static Singleton getInstance() {
       if(instance == null) {
            synchronized(Singleton.class) {
                if(instance == null) {
                     instance = new Singleton();
                }
            }
       }
       return instance;
   }
}
```