# On Thread Synchronization in Java

Volker Werling, 1714549, <volker.werling@stud.hs-mannheim.de>

Hochschule Mannheim – Wintersemester 2018/2019 - PPR

Version 1, February 2019

# Contents

# Figures



# Listings

# 1 Preface

This work focuses on synchronization strategies for multithreading with Java, which are present in the Java Standard Libraries within the package `java.util.concurrent`. While Threads in Java are shortly introduced at the beginning basic knowledge about Java as a language and concurrency in general is expected. Each synchronization mechanism of Java and the Standard Library are described with a runnable code snippet to make the usage clear.

# 2 Threads

## 2.1 Introduction

Independent sequences of execution can be modelled as processes and threads. A main difference between both is that processes generally run in separate memory spaces, while threads share their memory space.

Processes as well as threads are modelled as objects in the Java Standard Libraries. While processes are also an important construct of concurrency in Java there is a much bigger focus on in-process threading within Java as a language and the Java Standard Libraries. Having multiple threads of a program run in parallel with shared memory space has big advantages as it makes communication easy and efficient. At the same time, it introduces hazards, namely *thread interference* and *memory consistency errors.* Synchronization is needed to tackle those problems, while again it may also introduce *thread contention* problems [1, p. Synchronization].

To make avoiding those hazards easier the Java language and the Standard Libraries provide several constructs which are described in the latter parts of this document.

### Thread usage

A thread can easily be obtained by creating a new `Thread` object. The operations to execute within this thread are provided as a `Runnable` object. Calling `Thread.start()` leads to the execution of the provided `Runnable` object.

In fact, every program running on a JVM also runs in a special thread, the so-called *main thread*.

A Java program may run as many threads as it likes but performance-wise it makes sense to limit the amount of concurrently running threads. In general, it is advised to only run as many threads in parallel as are physically supported by the operating system to avoid decreased performance because of overhead introduced by scheduling of threads. The amount of supported threads is accessible via `Runtime.getRuntime().availableProcessor()`. Also, while threads may be created via the Thread object Java also has more sophisticated utility classes for thread management, so called `Executors`, which should be used in applications more heavily focused on multithreading.

## Thread states

A thread in Java can enter different states which are denoted in Figure 2-1. Upon creation it automatically enters the state READY, from where it can enter the RUNNING state by beeing startet with `Thread.start()`. Naturally upon completion of execution a thread enters the TERMINATED state, from where it cannot be restarted. While the thread itself cannot be reused, the runnable can be assigned to another thread which in can again be started.

Upon different synchronization blocks the thread enters different states:

*BLOCKED.* This state is entered, when a thread has to wait at a synchronization point denoted by synchronized. By itself this state is only left when the monitor becomes aquirable to the waiting thread. Naturally it can also be left by interrupting the thread at this point.

*WAITING.* The state WAITING is entered, when `Object.wait()` is called on a monitor. The calling thread then enters the waiting set of the monitor and may only leaf if upon notification of another active thread on the monitor.

*TIMED WAITING.* Timed waiting is entered when `Object.wait(timeout)` is called. This puts the thread in the waiting set of the monitor on which wait was called. In contrast to WAITING the thread automatically leaves the monitor's waiting set after the specified timeout. The state can as well be left earlier when `notify` was called on the monitor. The state may also be entered by calling `Thread.sleep(timeout)`. In this particular case the state is left after the timeout and only then.

For both waiting states (sans the `Thread.sleep(timeout)` speciality) a active monitor is necessary, so both can only be called inside a synchronized block, since otherwise there would be not waiting set for them to enter. This is further explained in the following section.

The TERMINATED state is reached upon completion of a threads operations whether as a result of succesful completion or failure of any kind.

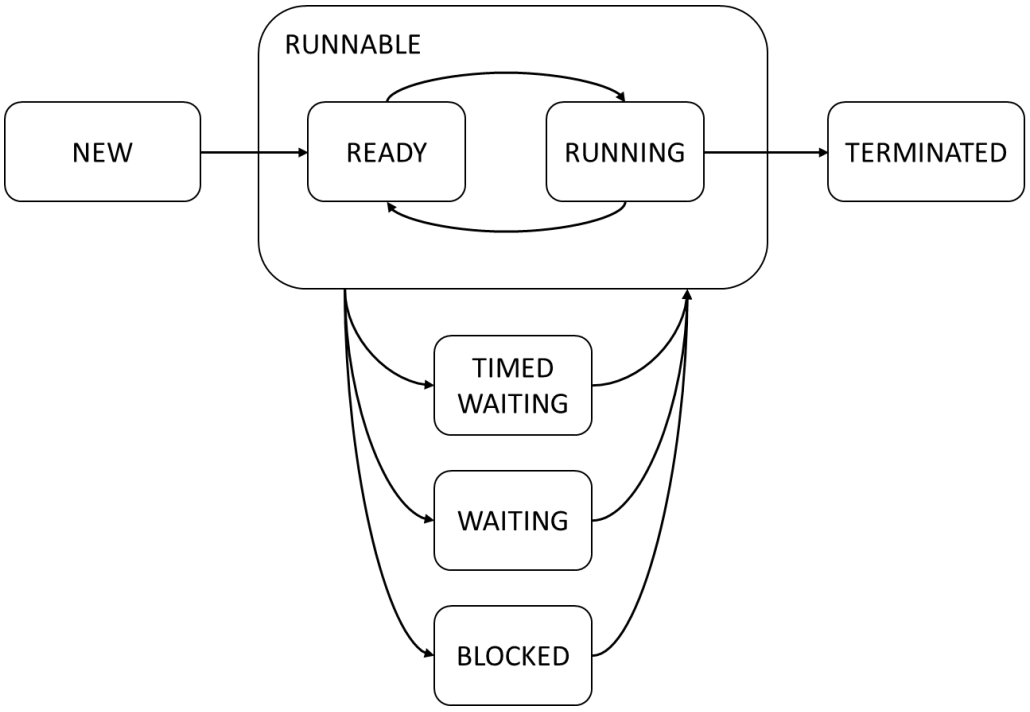


*Figure 2-1 Thread states*

## 2.2 Synchronization

For thread synchronization the Java language specification offers the keyword `synchronized` [2, p. Synchronization]. It is implemented using so called monitors. Each created object is associated with such a monitor which acts as a lock, that can be locked or unlocked by a thread. The keyword can be used to guard a method or a block of code. In both cases the lock is automatically acquired by a thread upon reaching the keyword and is again unlocked when reaching the end of the block or method.

### `synchronized` methods

An example of a synchronized method can be seen in Listing 2.1 Synchronization on a method. The static field counter is guarded by the synchronized method increment. The manipulation as well es the access to the counter field are guarded by the method. These guarantees, that upon printing the count the value of the field counter is still the same as it was after being changed in line 8 . Without the synchronization the field counter may have been incremented by another thread before the thread invoking the `printf` function is able to access the field value.

```
1.  class SynchronizedMethod
2.  {
3.    static int    counter = 0;
4.    static Object lock    = new Object();
5.
6.    static synchronized void increment()
7.    {
8.      counter = counter + 1;
9.      System.out.printf("Counter is %s%n", counter);
10.   }
11.
12.   public static void main(String[] args)
13.   {
14.     for (int i = 0; i < 10; i++)
15.     {
16.       new Thread(() -> increment()).start();
17.     }
18.   }
19. }
```

*Listing 2.1 Synchronization on a method*

An also important aspect is that synchronization on the method level not only blocks threads calling that method but also threads calling other `synchronized` methods of that object, since the monitor which guards the state is the object which has those methods defined itself.

### `synchronized` blocks

To have more fine-grained control over locking `synchronized` can also be defined on code blocks. In the following listing the same effect is achieved as with the method in the snippet before. The key difference is that now a monitor is “explicitly” defined. This is still called implicit locking since the interaction with the locking mechanism is still implicit. Synchronization on this level allows for different locks guarding different parts of a objects state. This can be used to improve throughput in contrast to synchronization with the objects own monitor.

```
1.  class SynchronizedBlock
2.  {
3.    static int    counter = 0;
4.    static Object lock    = new Object();
5.
6.    static void increment()
7.    {
8.      synchronized (lock)
9.      {
10.       counter = counter + 1;
11.       System.out.printf("Counter is %s%n", counter);
12.     }
13.   }
14.
15.   public static void main(String[] args)
16.   {
17.     for (int i = 0; i < 10; i++)
18.     {
19.       new Thread(() -> increment()).start();
20.     }
21.   }
22. }
```

*Listing 2.2 Synchronization on a code block*

The synchronized keyword only leads to threads being able to enter the BLOCKED state. Since resumption after the BLOCKED state is automatic this only offers limited usability. For more sophisticated synchronization there exists helper methods on Object which allow us to move threads into the WAITING state. A thread can be actively suspended, to be later notified to resume operation. Upon calling `wait()` it enters the so-called *waiting set* of a monitor and thus the state WAITING (`wait(timeout)` for the state TIMED WAITING respectively).

Another thing which has not yet been stated is the *re-entrancy* of locks used via `synchronized`. That means, that a thread that already holds the lock for a guarded code block may re-enter that same block. This may happen when a method is called recurrently.

### `wait()` and `notify()`

Listing 2.3 shows a situation where having a thread wait may be desirable. The object `lock` is guarding the state of the variable `slot`. The thread running `Taker` enters the WAITING state and the `locks` waiting set when the `slot` variable is not available. The thread running `Puter` makes an object available at the variable `slot` and lets the threads in the waiting set of `lock` know about that fact by calling `notifyAll()`. Both threads seam to acquire the same lock and thus `Taker` should have to wait forever, since the lock cannot be acquired by `Puter` at the same time. But that is not true, because upon entering the waiting set a thread surrenders ownership of the lock it had acquired. Upon resumption a thread needs to reobtain ownership of the lock and thus only one thread of the waiting set may resume operation.

```
1.  class WaitNotify
2.  {
3.    static Object lock  = new Object();
4.    static Object slot  = null;
5.
6.    static class Taker implements Runnable
7.    {
```

```
8.      @Override
9.      public void run()
10.     {
11.       try
12.       {
13.         synchronized (lock)
14.         {
15.           while (slot == null)
16.             lock.wait();
17.
18.           Object o = slot;
19.           slot = null;
20.           System.out.printf("Taken %s%n", o);
21.
22.           lock.notifyAll();
23.         }
24.       }
25.       catch (InterruptedException e)
26.       {
27.         return;
28.       }
29.     }
30.   }
31.
32.   static class Puter implements Runnable
33.   {
34.     @Override
35.     public void run()
36.     {
37.       try
38.       {
39.         synchronized (lock)
40.         {
41.           while (slot != null)
42.             lock.wait();
43.
44.           slot = new Object();
45.
46.           lock.notifyAll();
47.         }
48.       }
49.       catch (InterruptedException e)
50.       {
51.         return;
52.       }
53.     }
54.   }
55.
56.   public static void main(String[] args)
57.   {
58.     new Thread(new Taker()).start();
59.     sleep(2, TimeUnit.SECONDS);
60.     new Thread(new Puter()).start();
61.   }
62. }
```

*Listing 2.3 Synchronization with `Object.wait()` and `Object.notify()`*

# 3 Locks

Up to this point the locking mechanisms on monitors were *implicit.* The Java Standard Libraries also offer explicit locking mechanisms which can be found in the package `java.util.concurrent.lock`. They may be used the same way as implicit locks but also provide more sophisticated mechanisms to interact with a monitor and the threads at it. `ReentrantLock` is the most prominent example of an explicit lock in Java.

## 3.1 ReentrantLock

The `ReentrantLock` [3, p. class: ReentrantLock] usage shown in Listing 3.1 achieves the same effect as would be achieved by using `synchronized` on the block of code in between the `lock()` and `unlock()` methods or on the method `increment()` itself.

Alternatively, it is possible to only try to acquire the lock via `tryLock()`. As a `boolean` result the methods tells us whether directly acquiring that lock at that point is possible at all.

In addition to providing the locking mechanism we can query the ReentrantLock object for information about waiting threads via `getWaitingThreads(condition)`, `hasQueuedThread(thread)`, `isHeldByCurrentThread()` and more [3, p. class: ReentrantLock]. Information which is not in the same way easily accessible when using intrinsic locking.

Also, as with locks provided via the `synchronized` keyword, `ReentrantLock` guarantees re-entrancy of the lock, hence the name.

```
1.  class ReentrantLocking
2.  {
3.    static ReentrantLock  lock    = new ReentrantLock();
4.    static int            counter = 0;
5.
6.    static void increment()
7.    {
8.      lock.lock();
9.      System.out.printf("Waiting Threads: %s%n", lock.getQueueLength());
10.     sleep(1000, TimeUnit.MILLISECONDS);
11.     counter = counter + 1;
12.     System.out.printf("Incrementing at %s%n", Instant.now());
13.     lock.unlock();
14.   }
15.
16.   public static void main(String[] args)
17.   {
18.     Runnable incrementer = () -> increment();
19.     for (int i = 0; i < 10; i++)
20.     {
21.       new Thread(incrementer).start();
22.     }
23.   }
24. }
```

*Listing 3.1 Explicit Re-entrant Locking*

## 3.2 Condition

Further it is possible to create `Condition` [3, p. Condition] objects from the lock. A `Condition` factors out the `wait()` and `notify()` mechanisms introduced earlier and makes it more concise. Conditions are always bound to the lock they are created from. The usage can be seen in Listing 3.2 which reimplements Listing 2.3 while using a `ReentrantLock` and `Conditions`.

```
1.  class ConditionWaiting
2.  {
3.    static ReentrantLock  lock  = new ReentrantLock();
4.    static Condition      set   = lock.newCondition();
5.    static Condition      empty = lock.newCondition();
6.
7.    static Object slot;
8.
9.    static void put(Object o) throws InterruptedException
10.   {
11.     lock.lock();
12.
13.     if (slot != null)
14.       empty.await();
15.
16.     slot = o;
17.     set.signal();
18.
19.     lock.unlock();
20.   }
21.
22.   static Object take() throws InterruptedException
23.   {
24.     lock.lock();
25.     Object result;
26.
27.     if (slot == null)
28.       set.await();
29.     result = slot;
30.     slot = null;
31.
32.     empty.signal();
33.     lock.unlock();
34.     return result;
35.   }
36. }
```

*Listing 3.2 Object.wait() mechanism with Condition*

# 4 Synchronizers

Besides locks there are lots of other utility objects provided which can be described as synchronizers [4, p. 5.5 Synchronizers]. They coordinate the control flow of threads based on their state. The Java Standard Library contains several of those objects which help with more concise syntax for certain synchronization scenarios.

## 4.1 BlockingQueue

A `BlockingQueue` [3, p. class: BlockingQueue] is a construct which is helpful in Producer-Consumer situations. It extends the `Queue` interface which itself extends `Collection` making it a concurrency enabled one.

Since BlockingQueue is only an interface there are several implementations each with their own speciality. The example in Listing 4.1 uses a `SynchronousQueue` which as its speciality does not even have a capacity at all, which is counter intuitive for it being a collection. The methods used in the example are both blocking. The thread calling `take()` on the queue must wait for an object to become available. The same is true for the opposite case. A thread putting an object onto the queue via `put(object)` must wait until another thread comes along to take the value before the operation returns. As usual those methods are also available with timeouts to avoid having stalled threads.

It may also be noted that most of the methods introduced by the `Collection` interface have special semantics in contrast to the more widely known and used `Collection` implementations in the `java.util` package. It is recommended to consult the documentation on the usage of those special collections.

```
1.  class Queue
2.  {
3.    static BlockingQueue<Object> queue = new SynchronousQueue<>();
4.
5.    static void put(Object o)
6.    {
7.      try
8.      {
9.        queue.put(o);
10.     }
11.     catch (InterruptedException e)
12.     {
13.       return;
14.     }
15.   }
16.
17.   static Object take()
18.   {
19.     try
20.     {
21.       return queue.take();
22.     }
23.     catch (InterruptedException e)
24.     {
25.       return null;
26.     }
27.   }
28.
```

```
29.   public static void main(String[] args)
30.   {
31.     new Thread(() -> {
32.       System.out.printf("Taken %s", take());
33.     }).start();
34.
35.     sleep(5, TimeUnit.SECONDS);
36.
37.     new Thread(() -> {
38.       put(new Object());
39.     }).start();
40.   }
41. }
```

*Listing 4.1 Synchronization on a queue*

A `Queue` only supports adding objects to it from one end. There is also a two-ended variant called `Deque`, which also has a corresponding concurrent variant `BlockingDeque` [3, p. class: BlockingDeque] with similar semantics.

## 4.2 Exchanger

An `Exchanger` [3, p. Exchanger] marks a synchronization point upon which two threads can exchange information in between each other. It only offers one method, `exchange(object)` (as well as a timed variant, `exchange(object, timeout, unit)`) and offers no other semantics.

```
1.  class Exchanging
2.  {
3.    static Exchanger<Object> exchanger = new Exchanger<>();
4.
5.    static Thread producer = new Thread(() -> {
6.      sleep(5, TimeUnit.SECONDS);
7.      exchange(new Object());
8.    });
9.
10.   static Thread consumer = new Thread(() -> {
11.     System.out.println(exchange(null));
12.   });
13.
14.   private static Object exchange(Object o)
15.   {
16.     try
17.     {
18.       return exchanger.exchange(o);
19.     }
20.     catch (InterruptedException e)
21.     {
22.       return null;
23.     }
24.   }
25.
26.   public static void main(String[] args)
27.   {
28.     producer.start();
29.     consumer.start();
30.   }
31. }
```

*Listing 4.2 Exchanging objects between Threads via an Exchanger*

## 4.3 CountDownLatch

The *CountDownLatch* [3, p. class: CountDownLatch] aids in synchronizing threads on a fixed count. Threads may wait on such a gate and may only proceed if the count held by the latch is completely reduced. The example in Listing 4.3 has the main thread waiting for the worker threads to increment a counter. Only after each worker had its turn incrementing and counting down on the latch via `countdown()` the main thread waiting on the latch may proceed and print out the accumulated count.

So, the `CountDownLatch` can either be used to synchronize a fixed amount of threads or have an operation be executed for a fixed amount of times. After counting down to zero the latch is spent and may not be reused. To have a repeatable latch one may use a `CyclicBarrier` (see 4.4) although that may only be possible in special cases since semantics differ.

As usual there is also a timed variant, `countdown(timeout, unit)`.

```
1.  class CountDownLatchTest
2.  {
3.    static CountDownLatch latch;
4.    static AtomicInteger  aInt  = new AtomicInteger();
5.
6.    static class Worker implements Runnable
7.    {
8.      private int number;
9.
10.     public Worker(int number)
11.     {
12.       this.number = number;
13.     }
14.
15.     @Override
16.     public void run()
17.     {
18.       System.out.printf("Worker %s: %s%n", number,
19.                                             aInt.incrementAndGet());
20.       latch.countDown();
21.     }
22.   }
23.
24.   public static void main(String[] args)
25.   {
26.     latch = new CountDownLatch(5);
27.
28.     IntStream.of(1, 2, 3, 4, 5).forEach(i -> {
29.       Thread thread = new Thread(new Worker(i));
30.       thread.start();
31.     });
32.
33.     try
34.     {
35.       latch.await();
36.       System.out.printf("Result is %s", aInt.get());
37.     }
38.     catch (InterruptedException e)
39.     {
40.       return;
```

```
41.     }
42.   }
43. }
```

*Listing 4.3 Waiting for threads to finish with a latch*

## 4.4 CyclicBarrier

The *CyclicBarrier* [3, p. class: CyclicBarrier] marks a synchronization point upon which a set amount of threads need to arrive to allow all of them to proceed. In contrast to the `CountDownLatch` the barrier is reusable.

It additionally provides the possibility to run a barrier action which may be used to update a shared state used by the threads synchronizing on the barrier. The threads waiting on the barrier may only resume after the barrier action has terminated as can be seen in Listing 4.4. If it wasn't for the boolean field `done` be set to `true` by the barrier action the worker threads would never terminate.

```
1.  class CyclicBarrierIncrementer
2.  {
3.    static CyclicBarrier barrier;
4.    static boolean  done  = false;
5.
6.    static AtomicInteger aInt = new AtomicInteger();
7.
8.    static class Worker implements Runnable
9.    {
10.     private int number;
11.
12.     public Worker(int number)
13.     {
14.       this.number = number;
15.     }
16.
17.     @Override
18.     public void run()
19.     {
20.       while (!done)
21.       {
22.         System.out.printf("Worker %s: %s%n",
23.                           number,
24.                           aInt.incrementAndGet());
25.         try
26.         {
27.           barrier.await();
28.           System.out.printf(
29.                             "Barrier passed. Worker %s free.%n",
30.                             number);
31.         }
32.         catch (InterruptedException | BrokenBarrierException e)
33.         {
34.           return;
35.         }
36.       }
37.     }
38.   }
39.
40.   public static void main(String[] args)
41.       throws InterruptedException
```

```
42.   {
43.     List<Thread> threads = new ArrayList<>();
44.
45.     barrier = new CyclicBarrier(5, () -> {
46.       System.out.println(aInt.get());
47.       done = true;
48.     });
49.
50.     IntStream.of(1, 2, 3, 4, 5).forEach(i -> {
51.       Thread thread = new Thread(new Worker(i));
52.       threads.add(thread);
53.       thread.start();
54.     });
55.
56.     for (Thread thread : threads)
57.     {
58.       thread.join();
59.     }
60.   }
61. }
```

*Listing 4.4 Barrier Synchronization*

## 4.5 Phaser

Much like `CyclicBarrier` `Phaser` [3, p. class: Phaser] allows to build logic where multiple threads must wait on a barrier before they may progress. In contrast to `CyclicBarrier` there is no need define a fixed number of threads (although it is also possible) for which to wait for. Instead the threads which wish to synchronize upon a phase register themselves to Phaser by calling register() on it. Upon repeated waiting threads progress together in so called phases. The number of the current phase is returned when arriving at the barrier. When all registered threads arrive at a barrier execution may resume.

```
1.  class PhaserTest
2.  {
3.    static Phaser phaser;
4.
5.    static AtomicInteger aInt = new AtomicInteger();
6.
7.    static class Waiter implements Runnable
8.    {
9.      private int number;
10.
11.     public Waiter(int number)
12.     {
13.       this.number = number;
14.       phaser.register();
15.     }
16.
17.     @Override
18.     public void run()
19.     {
20.       System.out.printf("Waiter %s awaiting advance%n", number);
21.       aInt.incrementAndGet();
22.       int phase = phaser.arriveAndAwaitAdvance();
23.       System.out.printf("Waiter %s advanced and "
24.                         + "arriving on phase %s.%n",
```

```
25.                         number,
26.                         phase);
27.       aInt.incrementAndGet();
28.       phase = phaser.arriveAndAwaitAdvance();
29.       System.out.printf("Waiter %s advanced and "
30.                         + "arriving on phase %s.%n",
31.                         number,
32.                         phase);
33.       phaser.arriveAndDeregister();
34.       System.out.printf("Waiter %s arrived!%n", number);
35.     }
36.   }
37.
38.   public static void main(String[] args)
39.   {
40.     phaser = new Phaser() {
41.       @Override
42.       protected boolean onAdvance(int phase, int registeredParties)
43.       {
44.         if (registeredParties == 0)
45.         {
46.           System.out.printf("Count is %s%n", aInt.get());
47.           return aInt.get() == 10;
48.         }
49.         return false;
50.       }
51.     };
52.     IntStream.of(1, 2, 3, 4, 5).forEach(i -> {
53.       Thread thread = new Thread(new Waiter(i));
54.       thread.start();
55.     });
56.   }
57. }
```

*Listing 4.5 Synchronization on phase gates*

## 4.6 Semaphore

A `Semaphore` [3, p. class: Semaphore] is a barrier that restricts access and is often used to restrict access to a resource. This is achieved by `Semaphore` by holding a counter about the number of handed out permits for accessing the restricted area. There is not necessarily a 1:1 relationship between an accessing thread and a permit. A caller may as well acquire more than one permit at the semaphore barrier. When trying to acquire a permit via `acquire()` at thread enters the BLOCKING state if there is none available at that moment. Upon leaving the restricted area a permit is handed back to the semaphore by calling `release()` which in turn allows blocking threads to pass the barrier by acquiring a permit.

The example in Listing 4.6 shows a situation in which only two threads at a time can access the guarded `printf` statement since the it was created as `new Semaphore(2)`. The acquisition of a permit from the semaphore is inherently unfair. That means that there is no guarantee for the blocked threads to acquire the next free permit in the order they arrived. To grant that guarantee a semaphore must be created with `fairness` set to true – `new Semaphore(count, true)`. This should be noted as a mechanism to avoid *thread starvation*.

When creating the semaphore with only one permit it essentially acts as a lock. But in contrast to a lock as created by the `synchronized` keyword a semaphore lock may also be released by another thread, and not only by that thread which has acquired it.

As with `ReentrantLock` earlier it is also possible to probe the semaphore for acquisition of permit via `tryAcquire()`. It is important to note that this acquisition attempt will not honour the fairness setting at all, if a permit is available it is taken, at any time.

```
1.  class SemaphoreRateLimiting
2.  {
3.    static Semaphore semaphore;
4.
5.    static class Worker implements Runnable
6.    {
7.      private int number;
8.
9.      public Worker(int number)
10.     {
11.       this.number = number;
12.     }
13.
14.     @Override
15.     public void run()
16.     {
17.       try
18.       {
19.         semaphore.acquire();
20.         System.out.printf("Worker %s aquired semaphore at %s.%n",
21.                           number,
22.                           Instant.now());
23.         Thread.sleep(2000);
24.         semaphore.release();
25.       }
26.       catch (InterruptedException e)
27.       {
28.         return;
29.       }
30.     }
31.   }
32.
33.   public static void main(String[] args)
34.   {
35.     semaphore = new Semaphore(2);
36.
37.     IntStream.of(1, 2, 3, 4, 5, 6, 7, 8).forEach(i -> {
38.       Thread thread = new Thread(new Worker(i));
39.       thread.start();
40.     });
41.   }
```

*Listing 4.6 Rate limiting with a Semaphore*

## 4.7 FutureTask

Albeit not es specialized as the synchronization objects prior `FutureTask` [3, p. class: Semaphore] also is a kind of synchronization barrier. `FutureTask` is an implementation of a runnable which makes it possible to get its execution result. Since the execution may finish at any point asynchronously a thread

that wants to access the computations result has to wait for it to finish, effectively building a barrier for that waiting thread. This happens when calling `get()` (or the timing out version `get(timout, unit)`). An example of this is shown in Listing 4.7.

```
1.  class Future
2.  {
3.    public static void main(String[] args)
4.                                    throws InterruptedException,
5.                                           ExecutionException
6.    {
7.      FutureTask<?> task = new FutureTask<>(() -> {
8.        sleep(2, TimeUnit.SECONDS);
9.        return new Object();
10.     });
11.
12.     new Thread(task).start();
13.
14.     task.get();
15.   }
16. }
```

*Listing 4.7 Waiting on a FutureTask*

# 5 Supplementary Notes

All code snippets present in this document are complete and can be run with a standard Java 8 installation by copying and pasting them. The snippets aim to be self-sustainable in conveying the usage of the introduced concept without further explanations in text form by using prints to the console and sleeps where necessary.

Imports are missing but can be easily inserted with any standard Java IDE like Eclipse or IntelliJ.

The code snippets can also be accessed as a Gradle project on GitHub at https://github.com/volkerw00/hsma_ppr_java_thread_synchronization.

In some code snippets a helper method named *sleep* is used which is the following:

```
1.  public static void sleep(int time, TimeUnit unit)
2.  {
3.    try
4.    {
5.      unit.sleep(time);
6.    }
7.    catch (InterruptedException e)
8.    {
9.      return;
10.   }
11. }
```

*Listing 5.1 sleep helper method*

# 6 References

[1] Oracle, “The Java™ Tutorials - Lesson: Concurrency,” Oracle, [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html. [Accessed February 2019].

[2] Oracle, “Java Language Specification - Chapter 17. Threads and Locks,” Oracle, [Online]. Available: https://docs.oracle.com/javase/specs/jls/se11/html/jls-17.html. [Accessed February 2019].

[3] Oracle, “JDK 11 Documentation - Package java.util.concurrent,” 2018. [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/package-summary.html. [Accessed February 2019].

[4] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes and D. Lea, Java Concurrency in Practice, Amsterdam: Addison-Wesley Longman, 2006.