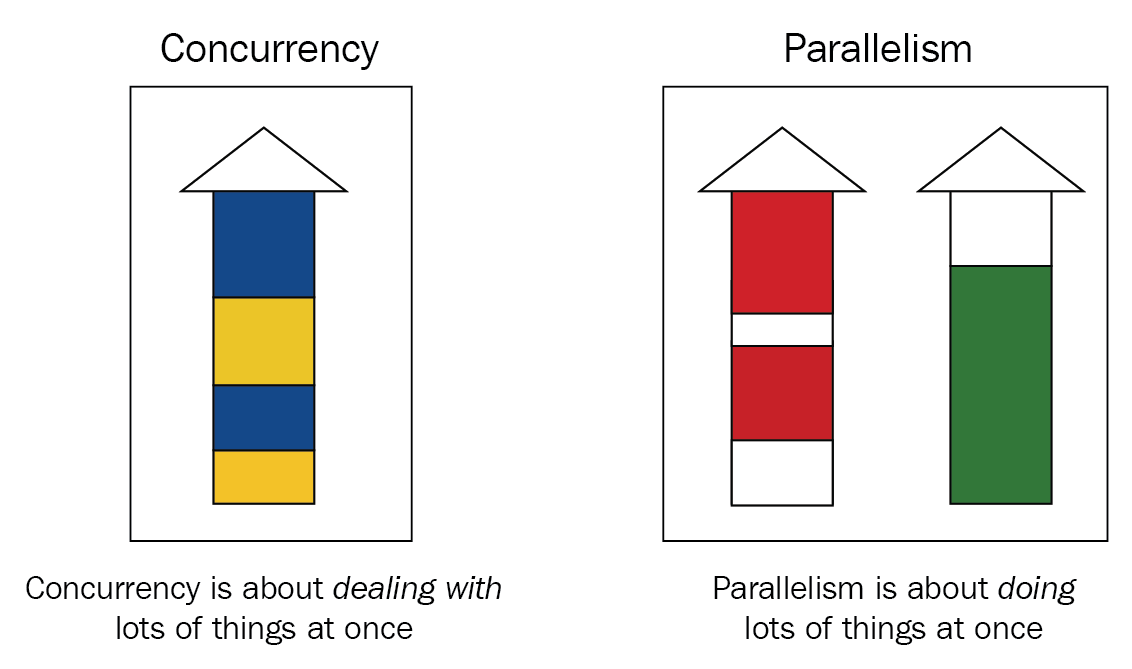
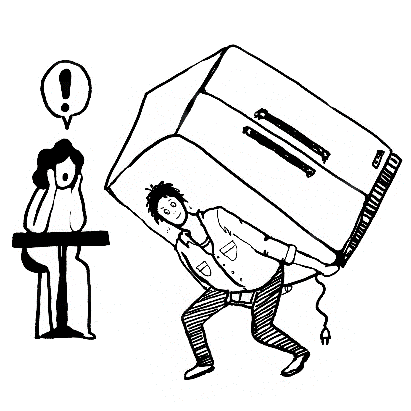
**CONCURRENCY:**  

Concurrency is the ability to run several programs or several parts of a program in parallel. If a time consuming task can be performed asynchronously or in parallel, this improve the throughput and the interactivity of the program.



**1. Overview**

The*java.util.concurrent* package provides tools for creating concurrent applications.



**2. Main Components**

The*java.util.concurrent* contains way too many features to discuss in a single write-up. In this article, we will mainly focus on some of the most useful utilities from this package like:

* *Executor*
* *ExecutorService*
* *ScheduledExecutorService*
* *Future*
* *CountDownLatch*
* *CyclicBarrier*
* *Semaphore*
* *ThreadFactory*
* *BlockingQueue*
* *DelayQueue*
* *Locks*
* *Phaser*

### **2.1.**Executor

[***Executor***](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html)**is an interface that represents an object that executes provided tasks.**

It depends on the particular implementation (from where the invocation is initiated) if the task should be run on a new or current thread. Hence, using this interface, we can decouple the task execution flow from the actual task execution mechanism.

One point to note here is that Executor does not strictly require the task execution to be asynchronous. In the simplest case, an executor can invoke the submitted task instantly in the invoking thread.

We need to create an invoker to create the executor instance:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Invoker implements Executor {      @Override      public void execute(Runnable r) {          r.run();      }  } |

Now, we can use this invoker to execute the task.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public void execute() {      Executor executor = new Invoker();      executor.execute( () -> {          // task to be performed      });  } |

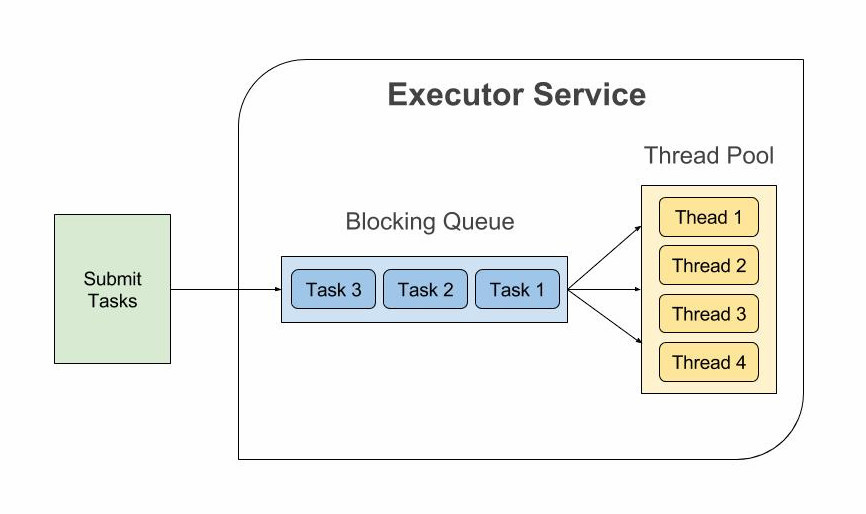
Point to note here is that if the executor can’t accept the task for execution, it will throw *[RejectedExecutionException](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RejectedExecutionException.html)*.

### **2.2.**ExecutorService

ExecutorService is a complete solution for asynchronous processing. It manages an in-memory queue and schedules submitted tasks based on thread availability.

To use ExecutorService, we need to create one Runnable class.

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class Task implements Runnable {      @Override      public void run() {          // task details      } |



Now we can create the ExecutorService instance and assign this task. At the time of creation, we need to specify the thread-pool size.

|  |  |
| --- | --- |
| 1 | ExecutorService executor = Executors.newFixedThreadPool(10); |

If we want to create a single-threaded ExecutorService instance, we can use **newSingleThreadExecutor(ThreadFactory threadFactory)** to create the instance.

Once the executor is created, we can use it to submit the task.

|  |  |
| --- | --- |
| 1  2  3 | public void execute() {      executor.submit(new Task());  } |

We can also create the Runnable instance while submitting the task.

|  |  |
| --- | --- |
| 1  2  3 | executor.submit(() -> {      new Task();  }); |

It also comes with two out-of-the-box execution termination methods. The first one is shutdown(); it waits till the all submitted task finish executing. The other method is shutdownNow() which immediately terminates all the pending/executing tasks.

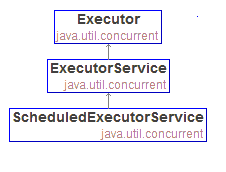
There is also another method awaitTermination(long timeout, TimeUnit unit) which forcefully blocks until all tasks have completed execution after a shutdown event triggered or execution-timeout occurred, or the execution thread itself is interrupted,

|  |  |
| --- | --- |
| 1  2  3  4  5 | try {      executor.awaitTermination( 20l, TimeUnit.NANOSECONDS );  } catch (InterruptedException e) {      e.printStackTrace();  } |

### **2.3.**ScheduledExecutorService

ScheduledExecutorService is a similar interface to ExecutorService, but it can perform tasks periodically.

**Executor and ExecutorService‘s methods are scheduled on the spot without introducing any artificial delay.** Zero or any negative value signifies that the request needs to be executed instantly.



We can use both Runnable and Callable interface to define the task.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public void execute() {      ScheduledExecutorService executorService        = Executors.newSingleThreadScheduledExecutor();        Future<String> future = executorService.schedule(() -> {          // ...          return "Hello world";      }, 1, TimeUnit.SECONDS);        ScheduledFuture<?> scheduledFuture = executorService.schedule(() -> {          // ...      }, 1, TimeUnit.SECONDS);        executorService.shutdown();  } |

ScheduledExecutorService can also schedule the task **after some given fixed delay**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | executorService.scheduleAtFixedRate(() -> {      // ...  }, 1, 10, TimeUnit.SECONDS);    executorService.scheduleWithFixedDelay(() -> {      // ...  }, 1, 10, TimeUnit.SECONDS); |

Here, the ***scheduleAtFixedRate( Runnable command, long initialDelay, long period, TimeUnit unit )*** method creates and executes a periodic action that is invoked firstly after the provided initial delay, and subsequently with the given period until the service instance shutdowns.

The ***scheduleWithFixedDelay( Runnable command, long initialDelay, long delay, TimeUnit unit )*** method creates and executes a periodic action that is invoked firstly after the provided initial delay, and repeatedly with the given delay between the termination of the executing one and the invocation of the next one.

### **2.4.**Future

**Future is used to represent the result of an asynchronous operation.**It comes with methods for checking if the asynchronous operation is completed or not, getting the computed result, etc.

What’s more, the cancel(boolean mayInterruptIfRunning) API cancels the operation and releases the executing thread. If the value of mayInterruptIfRunning is true, the thread executing the task will be terminated instantly.

Otherwise, in-progress tasks will be allowed to complete.

We can use below code snippet to create a future instance:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public void invoke() {      ExecutorService executorService = Executors.newFixedThreadPool(10);        Future<String> future = executorService.submit(() -> {          // ...          Thread.sleep(10000l);          return "Hello world";      });  } |

We can use following code snippet to check if the future result is ready and fetch the data if the computation is done:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | if (future.isDone() && !future.isCancelled()) {      try {          str = future.get();      } catch (InterruptedException | ExecutionException e) {          e.printStackTrace();      }  } |

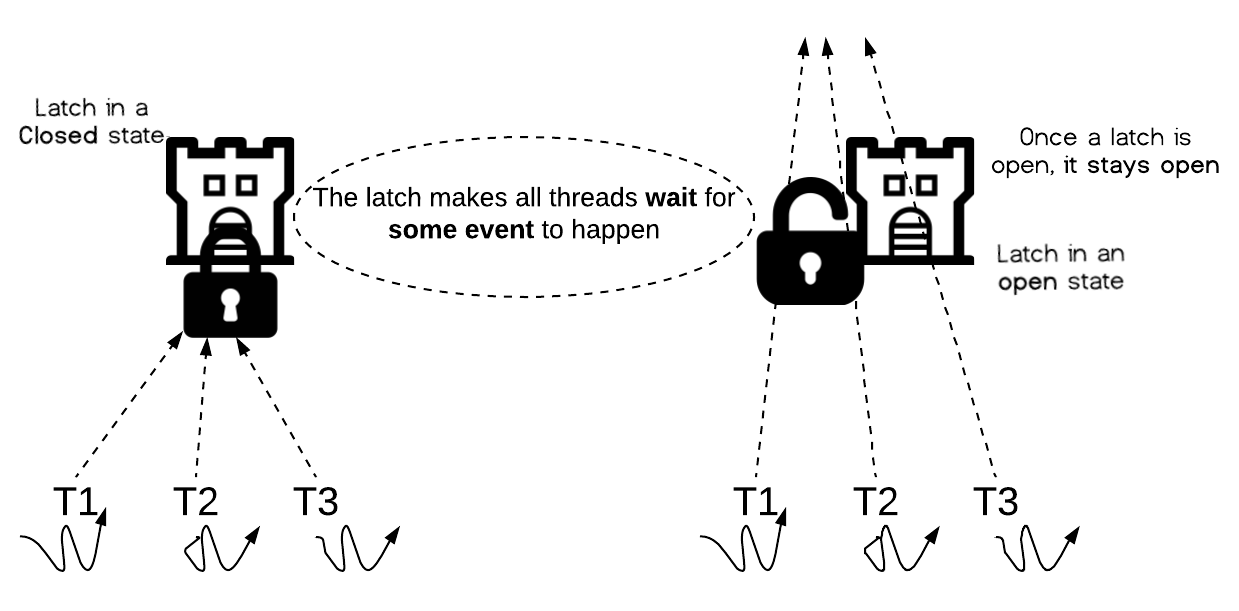
We can also specify a timeout for a given operation. If the task takes more than this time, a TimeoutException is thrown:

|  |  |
| --- | --- |
| 1  2  3  4  5 | try {      future.get(10, TimeUnit.SECONDS);  } catch (InterruptedException | ExecutionException | TimeoutException e) {      e.printStackTrace();  } |

### **2.5.**CountDownLatch

CountDownLatch (introduced in JDK 5) is a utility class which blocks a set of threads until some operation completes.

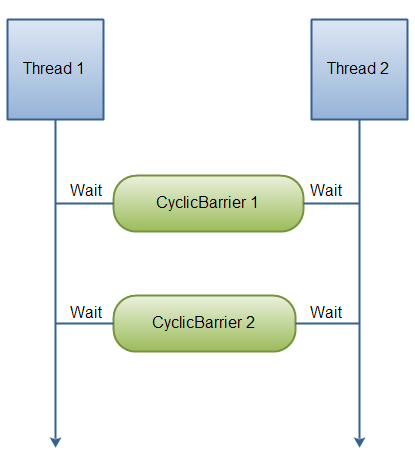
A CountDownLatch is initialized with a counter(Integer type); this counter decrements as the dependent threads complete execution. But once the counter reaches zero, other threads get released.

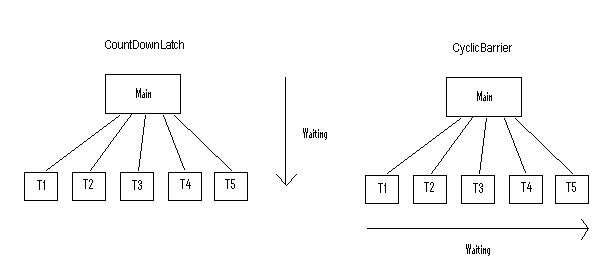


### **2.6.**CyclicBarrier

CyclicBarrier allows a number of threads to wait on each other, whereas CountDownLatch allows one or more threads to wait for a number of tasks to complete.

CyclicBarrier works almost same as CountDownLatch except that we can reuse it.



*Vs*

The first difference here is that the threads that are waiting are themselves the barrier.

Second, and more importantly, **the second *await()* is useless**.**A single thread can’t *count down*a barrier twice.**

Indeed, because *t* must *wait* for another thread to call *await()* – to bring the count to two – *t*‘s second call to *await()*won’t actually be invoked until the barrier is already broken!

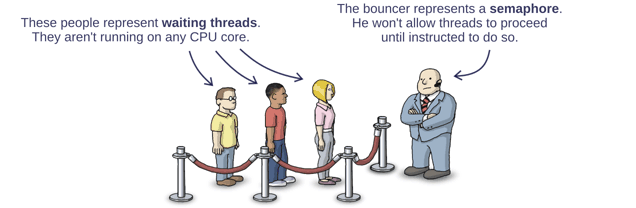
The second most evident difference between these two classes is reusability. To elaborate, **when the barrier trips in *CyclicBarrier*, the count resets to its original value.** ***CountDownLatch*is different because the count never resets.**

In the given code, we define a *CountDownLatch* with count 7 and count it through 20 different calls:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | CountDownLatch countDownLatch = new CountDownLatch(7);  ExecutorService es = Executors.newFixedThreadPool(20);  for (int i = 0; i < 20; i++) {      es.execute(() -> {          long prevValue = countDownLatch.getCount();          countDownLatch.countDown();          if (countDownLatch.getCount() != prevValue) {              outputScraper.add("Count Updated");          }      });  }  es.shutdown();    assertTrue(outputScraper.size() <= 7); |

### **2.7.**Semaphore

The Semaphore is used for blocking thread level access to some part of the physical or logical resource. A semaphore contains a set of permits; whenever a thread tries to enter the critical section, it needs to check the semaphore if a permit is available or not.



**If a permit is not available (via tryAcquire()), the thread is not allowed to jump into the critical section; however, if the permit is available the access is granted, and the permit counter decreases.**

Once the executing thread releases the critical section, again the permit counter increases (done by release()method).

We can specify a timeout for acquiring access by using the tryAcquire(long timeout, TimeUnit unit) method.

**We can also check the number of available permits or the number of threads waiting to acquire the semaphore.**

Following code snippet can be used to use implement a semaphore:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | static Semaphore semaphore = new Semaphore(10);    public void execute() throws InterruptedException {        LOG.info("Available permit : " + semaphore.availablePermits());      LOG.info("Number of threads waiting to acquire: " +        semaphore.getQueueLength());        if (semaphore.tryAcquire()) {          semaphore.acquire();          // ...          semaphore.release();      }    } |

### **2.8.**ThreadFactory

As the name suggests, ThreadFactory acts as a thread (non-existing) pool which creates a new thread on demand. It eliminates the need of a lot of boilerplate coding for implementing efficient thread creation mechanisms.

We can define a ThreadFactory:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class BaeldungThreadFactory implements ThreadFactory {      private int threadId;      private String name;        public BaeldungThreadFactory(String name) {          threadId = 1;          this.name = name;      }        @Override      public Thread newThread(Runnable r) {          Thread t = new Thread(r, name + "-Thread\_" + threadId);          LOG.info("created new thread with id : " + threadId +              " and name : " + t.getName());          threadId++;          return t;      }  } |

We can use this newThread(Runnable r) method to create a new thread at runtime:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | BaeldungThreadFactory factory = new BaeldungThreadFactory(      "BaeldungThreadFactory");  for (int i = 0; i < 10; i++) {      Thread t = factory.newThread(new Task());      t.start();  } |

### **2.9.**BlockingQueue

In asynchronous programming, one of the most common integration patterns is the [producer-consumer pattern](https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem). The java.util.concurrent package comes with a data-structure know as BlockingQueue – which can be very useful in these async scenarios.



## BlockingQueue****Types****

We can distinguish two types of BlockingQueue:

* unbounded queue – can grow almost indefinitely
* bounded queue – with maximal capacity defined

### **2.1. Unbounded Queue**

Creating unbounded queues is simple:

|  |  |
| --- | --- |
| 1 | BlockingQueue<String> blockingQueue = new LinkedBlockingDeque<>(); |

The Capacity of blockingQueue will be set to Integer.MAX\_VALUE. All operations that add an element to the unbounded queue will never block, thus it could grow to a very large size.

The most important thing when designing a producer-consumer program using unbounded BlockingQueue is that consumers should be able to consume messages as quickly as producers are adding messages to the queue. Otherwise, the memory could fill up and we would get an OutOfMemory exception.

### **2.2. Bounded Queue**

The second type of queues is the bounded queue. We can create such queues by passing the capacity as an argument to a constructor:

|  |  |
| --- | --- |
| 1 | BlockingQueue<String> blockingQueue = new LinkedBlockingDeque<>(10); |

Here we have a blockingQueue that has a capacity equal to 10. It means that when a consumer tries to add an element to an already full queue, depending on a method that was used to add it (offer(), add() or put()), it will block until space for inserting object becomes available. Otherwise, the operations will fail.

Using bounded queue is a good way to design concurrent programs because when we insert an element to an already full queue, that operations need to wait until consumers catch up and make some space available in the queue. It gives us throttling without any effort on our part.

## ****3.****BlockingQueue****API****

There are two types of methods in the BlockingQueue interface – methods responsible for adding elements to a queue and methods that retrieve those elements. Each method from those two groups behaves differently in case the queue is full/empty.

### **3.1. Adding Elements**

* add() – returns true if insertion was successful, otherwise throws an IllegalStateException
* put() – inserts the specified element into a queue, waiting for a free slot if necessary
* offer() – returns true if insertion was successful, otherwise false
* offer(E e, long timeout, TimeUnit unit) – tries to insert element into a queue and waits for an available slot within a specified timeout

### **3.2. Retrieving Elements**

* take() – waits for a head element of a queue and removes it. If the queue is empty, it blocks and waits for an element to become available
* poll(long timeout, TimeUnit unit) – retrieves and removes the head of the queue, waiting up to the specified wait time if necessary for an element to become available. Returns null after a timeout

### **2.10.**DelayQueue

DelayQueue is an infinite-size blocking queue of elements where an element can only be pulled if it’s expiration time (known as user defined delay) is completed. Hence, the topmost element (head) will have the most amount delay and it will be polled last.

## ****Implementing****Delayed ****for Elements in the****DelayQueue

Each element we want to put into the DelayQueue needs to implement the [*Delayed*](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Delayed.html) interface. Let’s say that we want to create a DelayObject class. Instances of that class will be put into the DelayQueue.

We’ll pass the String data and delayInMilliseconds as and arguments to its constructor:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class DelayObject implements Delayed {      private String data;      private long startTime;        public DelayObject(String data, long delayInMilliseconds) {          this.data = data;          this.startTime = System.currentTimeMillis() + delayInMilliseconds;      } |

We are defining a startTime – this is a time when the element should be consumed from the queue. Next, we need to implement the getDelay() method – it should return the remaining delay associated with this object in the given time unit.

Therefore, we need to use the TimeUnit.convert() method to return the remaining delay in the proper TimeUnit:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Override  public long getDelay(TimeUnit unit) {      long diff = startTime - System.currentTimeMillis();      return unit.convert(diff, TimeUnit.MILLISECONDS);  } |

When the consumer tries to take an element from the queue, the DelayQueue will execute getDelay() to find out if that element is allowed to be returned from the queue. If the getDelay() method will return zero or a negative number, it means that it could be retrieved from the queue.

We also need to implement the compareTo() method, because the elements in the DelayQueue will be sorted according to the expiration time. The item that will expire first is kept at the head of the queue and the element with the highest expiration time is kept at the tail of the queue:

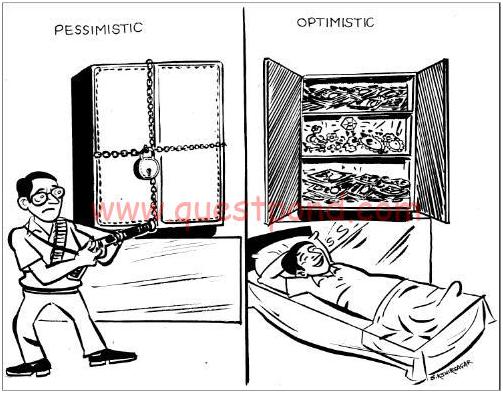
|  |  |
| --- | --- |
| 1  2  3  4  5 | @Override  public int compareTo(Delayed o) {      return Ints.saturatedCast(        this.startTime - ((DelayObject) o).startTime);  } |

### **2.11.**Locks

Not surprisingly, Lock is a utility for blocking other threads from accessing a certain segment of code, apart from the thread that’s executing it currently.



LOCK 



Simply put, a lock is a more flexible and sophisticated thread synchronization mechanism than the standard *synchronized* block.

The main difference between a Lock and a Synchronized block is that synchronized block is fully contained in a method; however, we can have Lock API’s lock() and unlock() operation in separate methods.

The*Lock*interface has been around since Java 1.5. It’s defined inside the *java.util.concurrent.lock*package and it provides extensive operations for locking.

In this article, we’ll explore different implementations of the *Lock* interface and their applications.

**2. Differences between *Lock* and *Synchronized* *block***

There are few differences between the use of synchronized *block* and using *Lock* API’s:

* **A *synchronized* *block* is fully contained within a method –**we can have *Lock* API’s *lock()* and *unlock()*operation in separate methods
* A s*ynchronized block* doesn’t support the fairness, any thread can acquire the lock once released, no preference can be specified. **We can achieve fairness within the *Lock* APIs by specifying the *fairness*property**. It makes sure that longest waiting thread is given access to the lock
* A thread gets blocked if it can’t get an access to the synchronized *block*. **The *Lock* API provides *tryLock()*method. The thread acquires lock only if it’s available and not held by any other thread.** This reduces blocking time of thread waiting for the lock
* A thread which is in “waiting” state to acquire the access to *synchronized block*, can’t be interrupted. **The *Lock*API provides a method *lockInterruptibly()*which can be used to interrupt the thread when it’s waiting for the lock**

**3. *Lock* API**

Let’s take a look at the methods in the *Lock* interface:

* ***void lock()****–*acquire the lock if it’s available; if the lock isn’t available a thread gets blocked until the lock is released
* ***void lockInterruptibly()*** – this is similar to the *lock(),*but it allows the blocked thread to be interrupted and resume the execution through a thrown *java.lang.InterruptedException*
* ***boolean tryLock()***– this is a non-blocking version of *lock()*method; it attempts to acquire the lock immediately, return true if locking succeeds
* ***boolean tryLock(long timeout, TimeUnit timeUnit)****–*this is similar to *tryLock(),*except it waits up the given timeout before giving up trying to acquire the *Lock*
* **void *unlock()*** – unlocks the *Lock* instance

A locked instance should always be unlocked to avoid deadlock condition. A recommended code block to use the lock should contain a *try/catch* and *finally* block:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Lock lock = ...;  lock.lock();  try {      // access to the shared resource  } finally {      lock.unlock();  } |

### **2.12.**Phaser

Phaser is a more flexible solution than CyclicBarrier and CountDownLatch – used to act as a reusable barrier on which the dynamic number of threads need to wait before continuing execution. We can coordinate multiple phases of execution, reusing a Phaser instance for each program phase.

## Phaser****API****

The Phaser allows us to build logic in which **threads need to wait on the barrier before going to the next step of execution**.

We can coordinate multiple phases of execution, reusing a Phaser instance for each program phase. Each phase can have a different number of threads waiting for advancing to another phase. We’ll have a look at an example of using phases later on.

To participate in the coordination, the thread needs to register() itself with the Phaser instance. Note that this only increases the number of registered parties, and we can’t check whether the current thread is registered – we’d have to subclass the implementation to supports this.

The thread signals that it arrived at the barrier by calling the arriveAndAwaitAdvance(), which is a blocking method. **When the number of arrived parties is equal to the number of registered parties, the execution of the program will continue**, and the phase number will increase. We can get the current phase number by calling the getPhase() method.

When the thread finishes its job, we should call the arriveAndDeregister() method to signal that the current thread should no longer be accounted for in this particular phase.

## ****3. Implementing Logic Using****Phaser ****API****

Let’s say that we want to coordinate multiple phases of actions. Three threads will process the first phase, and two threads will process the second phase.

We’ll create a LongRunningAction class that implements the Runnable interface:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class LongRunningAction implements Runnable {      private String threadName;      private Phaser ph;        LongRunningAction(String threadName, Phaser ph) {          this.threadName = threadName;          this.ph = ph;          ph.register();      }        @Override      public void run() {          ph.arriveAndAwaitAdvance();          try {              Thread.sleep(20);          } catch (InterruptedException e) {              e.printStackTrace();          }          ph.arriveAndDeregister();      }  } |

Atomic Classes:

Available from the Package “java.util.concurrent.atomic”.

List of Atomic Classes:

* AtomicBoolean - A boolean value that may be updated automatically.
* AtomicInteger - an int value that may be updated automatically.
* AtomicIntegerArray - an int array in which the elements are updated

automatically

* AtomicIntegerFieldUpdater<T> - A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes.
* AtomicLong – A long value that may be updated automatically.
* AtomicLongArray – A Long Array in which the elements are updated

Automatically.

* AtomicLongFieldUpdater<T> - A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes.
* AtomicMarkableReference<V> - An atomic markable reference maintains an object reference along with a mark bit,that can be updated automatically.
* AtomicReference<V> - An object reference that can be updated automatically.
* Atomic ReferenceArray<E> -cAn array of object references in which the elements are updated automatically.
* AtomicReferenceFieldUpdater<T> - A reflection-based utility that enables atomic updates to designated volatile reference fields of designated classes.
* AtomicStampedReference<V> - An atomic stamped reference maintains an object reference along with an integer stamp that can be updated automatically.

AtomicInteger atomicInt = new AtomicInteger(0);

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 1000)

.forEach(i -> executor.submit(atomicInt::incrementAndGet));

stop(executor);

System.out.println(atomicInt.get());

**THANKYOU**

**PRESENTED BY: (S S S): - SHAIK MOHAMMED RAFIQ**

**SAIPRIYA RAVIPALLY**

**SHRAVANI KOTHA**