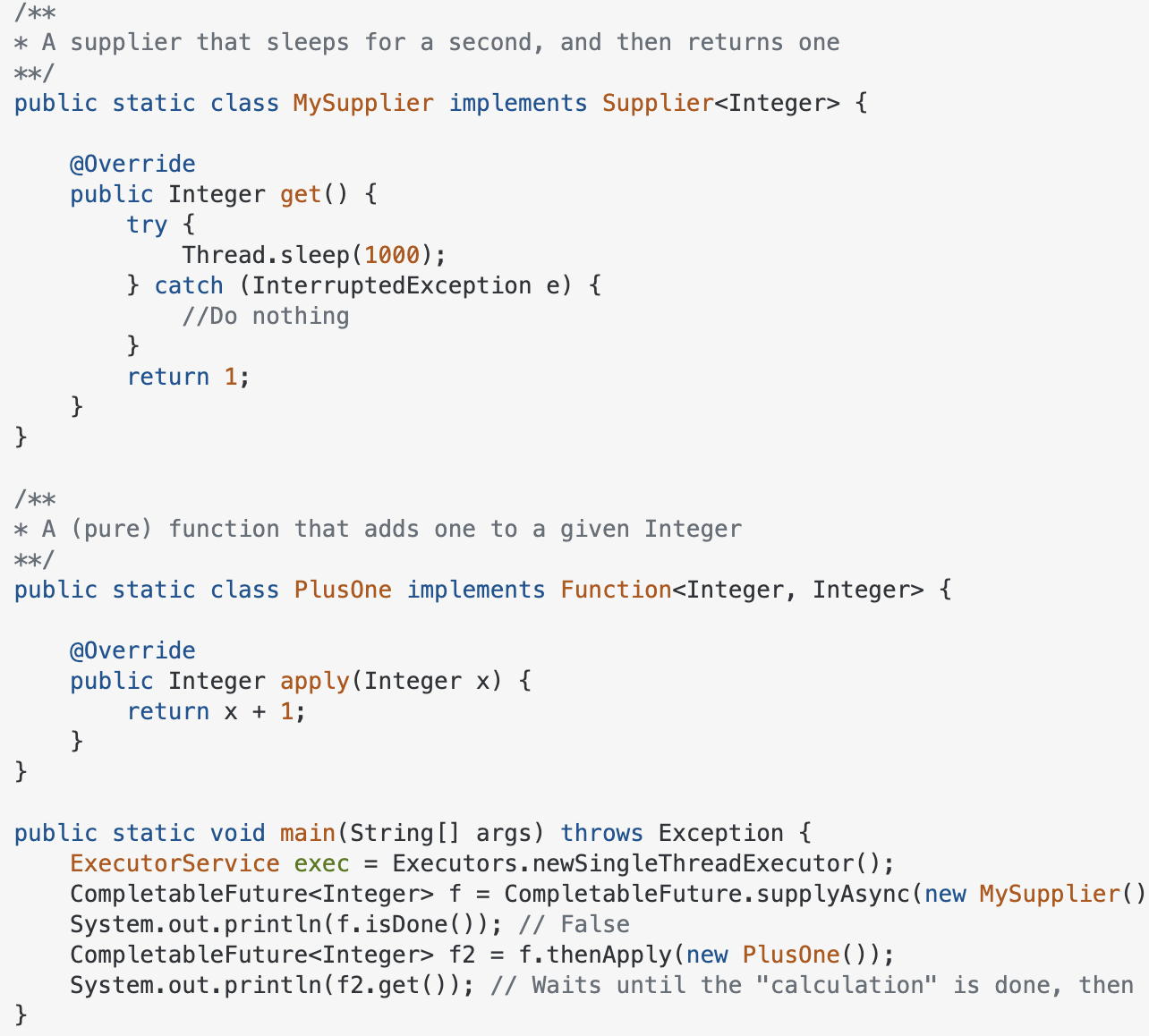
**Futures**

A Future is **used as a reference to the result of an asynchronous computation**. It provides an **isDone()** method to check **whether the computation is done or not**, and a **blocking get()** method to **retrieve the result of the computation** when it is done.



**CompletableFutures**

**CompletableFutures are Futures** that also **allow you to string tasks together in a chain**. You can use them to tell some worker thread to "go do some task X, and when you're done, go do this other thing using the result of X". **Using CompletableFutures**, you can **do something with the result of the operation without actually blocking a thread** to wait for the result.



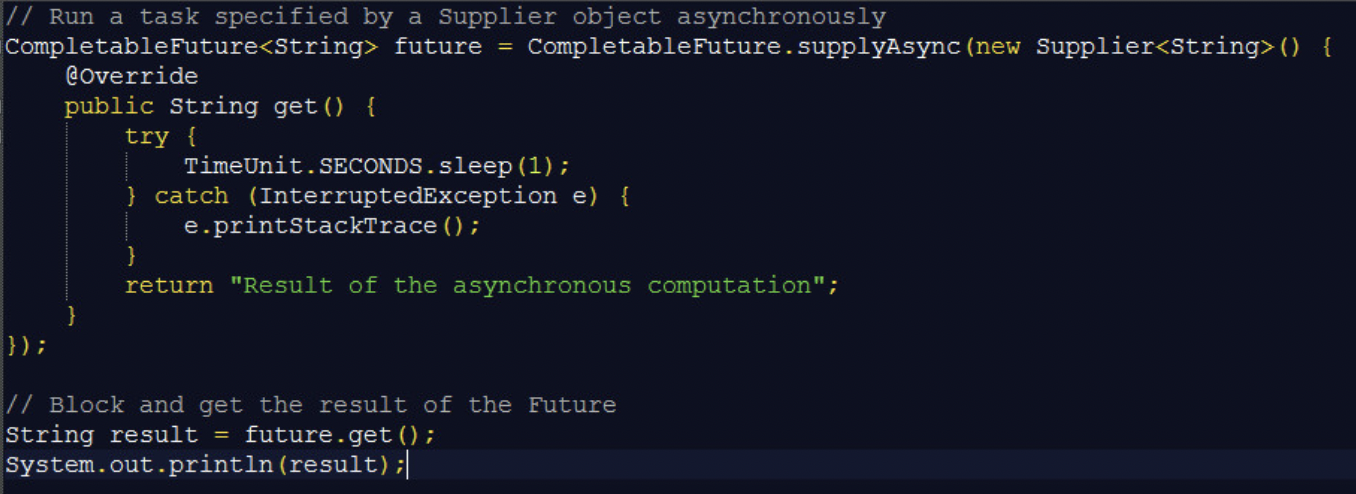
**Syntax:**

CompletableFuture<Integer> f = CompletableFuture.supplyAsync(new MySupplier(), exec);

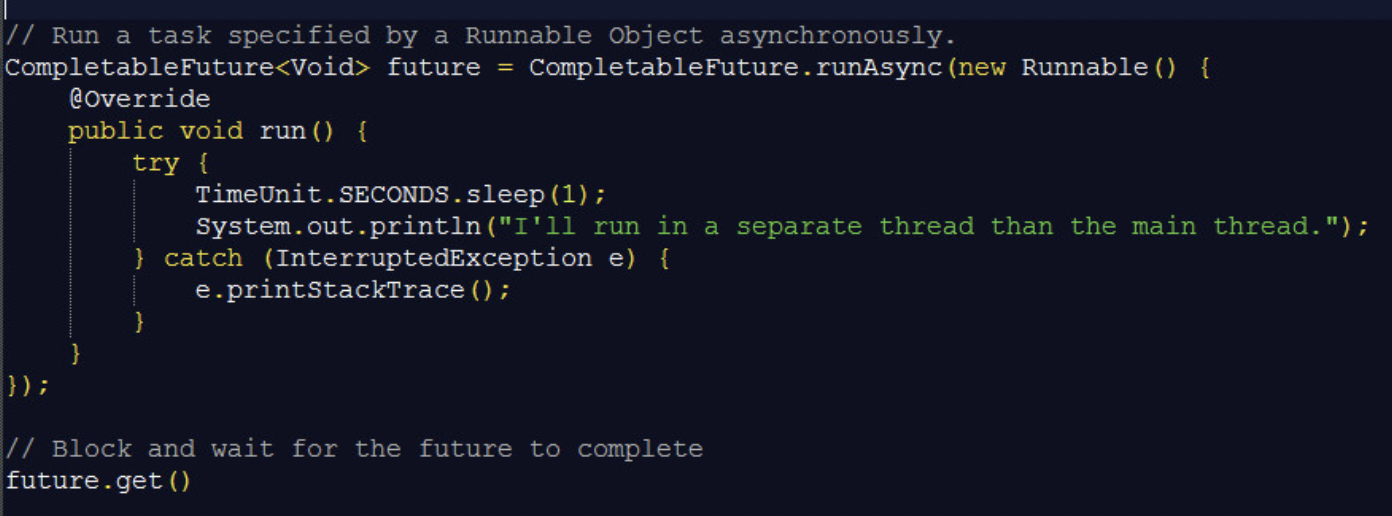
**Output:**

// Waits until the "calculation" is done, then prints 2

**CompletableFuture.supplyAsync()** **takes a** [**Supplier<T>**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html) and **returns CompletableFuture<T>** where T is the type of the value obtained by calling the given supplier. A **Supplier<T> is a simple functional interface that represents a supplier of results**. It has a **single get() method** where you can write your background task and return the result.



If you want to **run some background task asynchronously and don’t want to return anything from the task**, then you can use **CompletableFuture.runAsync()** method. It **takes a Runnable object** and **returns CompletableFuture<Void>**.



**Executors & Thread pools**

**Thread pool** is a core concept in multithreaded programming which, simply put, **represents a collection of idle threads that can be used to execute tasks**.

**Creating and starting a thread can be an expensive process**. By repeating this process every time we need to execute a task, we’re incurring a significant performance cost – which is exactly what we were attempting to improve by using thread pools.

**Java provides** its **own implementations of the thread pool pattern**, **through** objects called **executors**. These can be used through **executor interfaces** or **directly through thread pool implementations** – which does **allow for finer-grained control**.

The **java.util.concurrent package** contains the following interfaces:

* **Executor** – a simple interface for executing tasks
* **ExecutorService** – a more complex interface that contains additional methods for managing the tasks and executor itself
* **ScheduledExecutorService** – extends ExecutorService with methods for scheduling the execution of a task

The **Executors** **class** **contains factory methods for creating different types of thread pools**, while **Executor is the simplest thread pool interface**, with a **single execute() method**.



The **execute() method runs the statement if a worker thread is available**, or **places the Runnable task in a queue** to wait for a thread to become available.

The **factory methods** in the **Executors** **class** can create several types of thread pools:

* **newSingleThreadExecutor**() – single background thread with an unbounded queue
* **newFixedThreadPool**(int) – fixed-size thread pool that shares an unbounded queue
* **newCachedThreadPool**() – an unbounded thread pool, with automatic thread reclamation
* **newScheduledThreadPool**(int) – fixed-size thread pool with delayed & periodic task execution

One way to **create an ExecutorService** is to **use the factory methods from the Executors class**:



Besides the execute() method, the **ExecutorService** **interface** also defines a similar **submit()** method that can **return a Future object**:



As you can see in the example above, the Future interface can return the result of a task for Callable objects, and can also show the status of task execution.

The ExecutorService is not automatically destroyed when there are no tasks waiting to be executed, so to shut it down explicitly, you can use the shutdown() or shutdownNow() APIs:



The **ScheduledExecutorService** is a **subinterface of ExecutorService** – which adds methods for scheduling tasks:



The **schedule()** method **specifies a task to be executed**, **a delay value**, and **a TimeUnit for the value**:

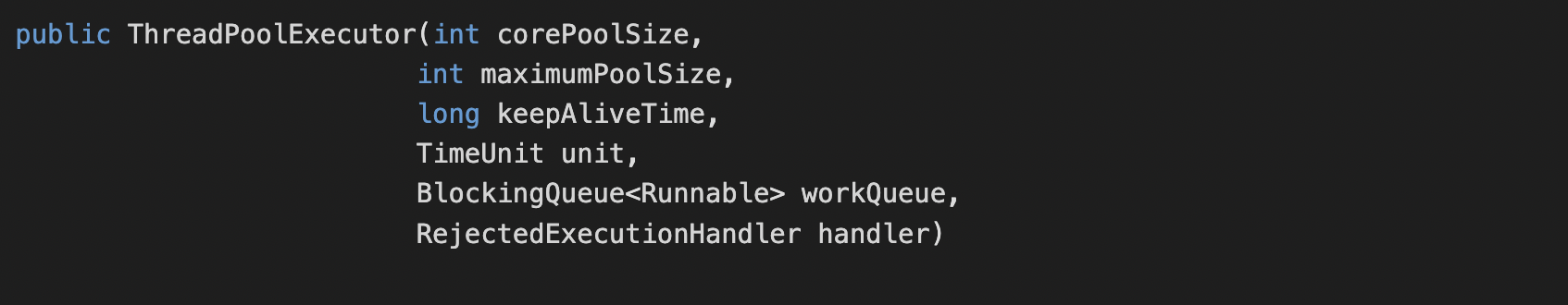


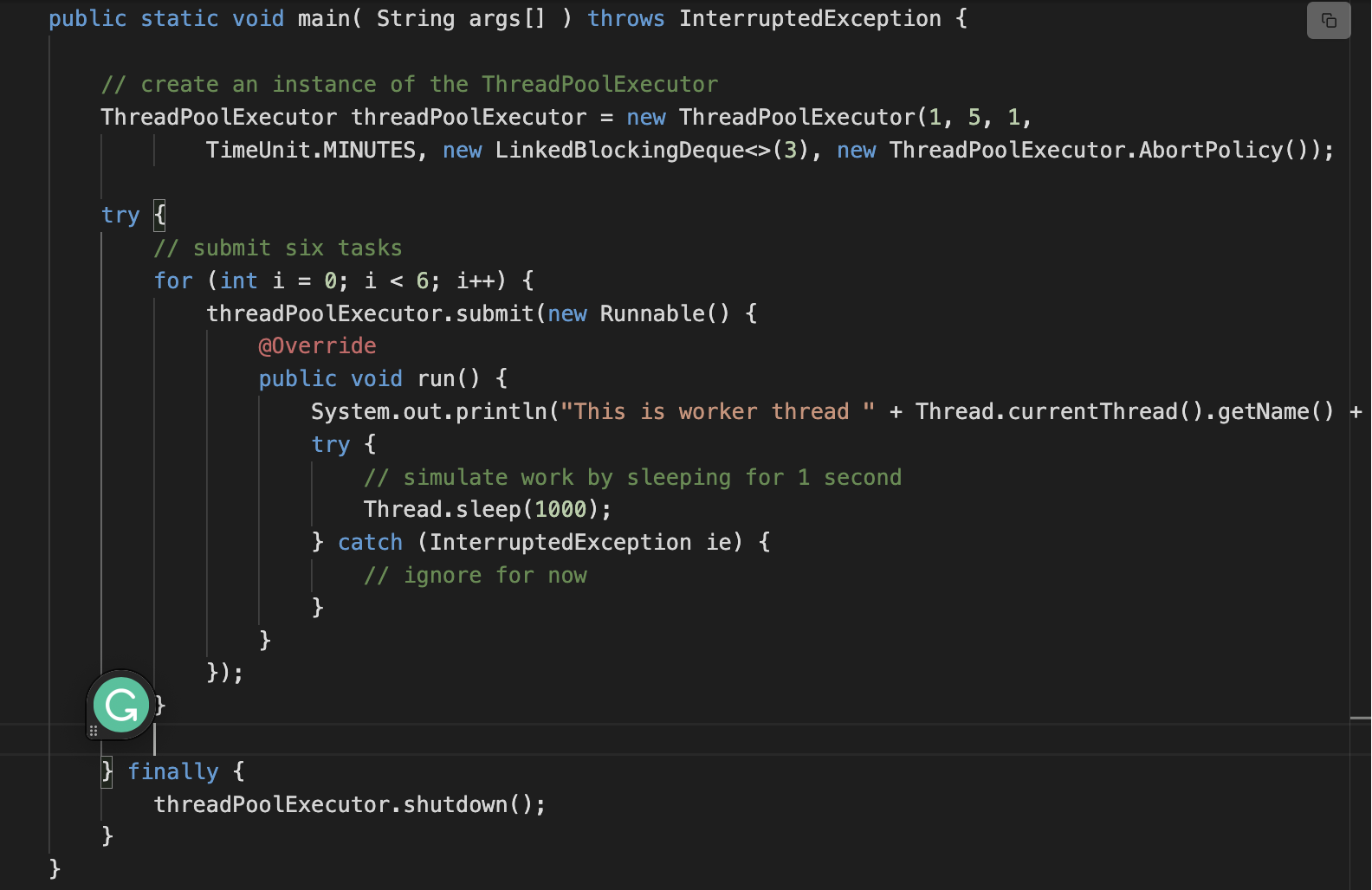
The **scheduleAtFixedRate()** method executes the task after a 2 ms delay, then repeats it every 2 seconds. Similarly, the **scheduleWithFixedDelay()** method starts the first execution after 2 ms, then repeats the task 2 seconds after the previous execution ends.



The **ThreadPoolExecutor** **implementation** **adds the ability to configure parameters, as well as extensibility hooks**.

Let’s consider the constructor that takes in the most arguments to instantiate the ThreadPoolExecutor class:





If we use a queue such as the **LinkedBlockingQueue** **without a predefined capacity**, the **queue can arbitrarily grow in size**. The consequence is that **tasks get added to the queue if all the corePoolSize threads are busy**. Interestingly, the **maximumPoolSize** **setting takes no effect** and **only corePoolSize threads are ever created**. Submitted tasks sit in the queue waiting for execution. Using this strategy we can see the **queue size grow indefinitely**.

We can also **define a capacity** when passing in the **LinkedBlockingQueue**. In that scenario, the **executor can reject newly submitted tasks if the queue has reached capacity** and **maximumPoolSize threads have been created** and are busy executing other tasks. **Note that with a defined capacity queue the setting maximumPoolSize becomes effective**.

If the **executor becomes overwhelmed with tasks**, it **can reject newly submitted tasks**. This occurs **when the executor has a defined maximum pool size and a defined queue capacity and both resources hit their limits**.

There are **four different policies** that can be **supplied to the executor** to determine the **course of action when tasks can’t be accepted** **anymore**.

1. **ThreadPoolExecutor.AbortPolicy** - The abort policy simply throws the runtime **RejectedExecutionException when a task can’t be accepted**.
2. **ThreadPoolExecutor.CallerRunsPolicy** - According to this policy the **thread invoking the execute() method of the executor itself runs the task**. This mechanism serves to **throttle the rate at which tasks are submitted** as the **submitting threads themselves end up executing the tasks** they submit.
3. **ThreadPoolExecutor.DiscardPolicy** - A **task that can’t be executed is simply dropped**.
4. **ThreadPoolExecutor.DiscardOldestPolicy** - When a task can’t be accepted for execution, this policy causes the **oldest unhandled request/task to be discarded** and then the **execution is retried for the just submitted task**.

Generally, **the use of the ThreadPoolExecutor class is discouraged** in the favor of thread pools that can be **instantiated using the Executors factory methods**.



However, the **ThreadPoolExecutor** **comes with several knobs and parameters** that can be **fine-tuned to suit unusual use-cases**.

**Blocking Queues**

A **blocking queue** is a **queue that blocks when you try to dequeue from it and the queue is empty**, **or if you try to enqueue items to it and the queue is already full**. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

**Reentrant Locks**

The **Java Lock interface**, **java.util.concurrent.locks.Lock**, **represents a concurrent lock that can be used to guard against race conditions** inside critical sections. Thus, the **Java Lock interface** **provides a more flexible alternative to a Java synchronized block**.

The main **differences between** a **Lock** anda **synchronized block** are:

1. **A synchronized block does not guarantee the sequence in which threads waiting to enter it are granted access**.
2. We **cannot pass any parameters** **to the entry of a synchronized block**. Thus, **having a** **timeout trying to get access to a synchronized block is not possible**.
3. The **synchronized block must be fully contained within a single method**. A **Lock** **can have its calls to lock()** and **unlock() in separate methods**.

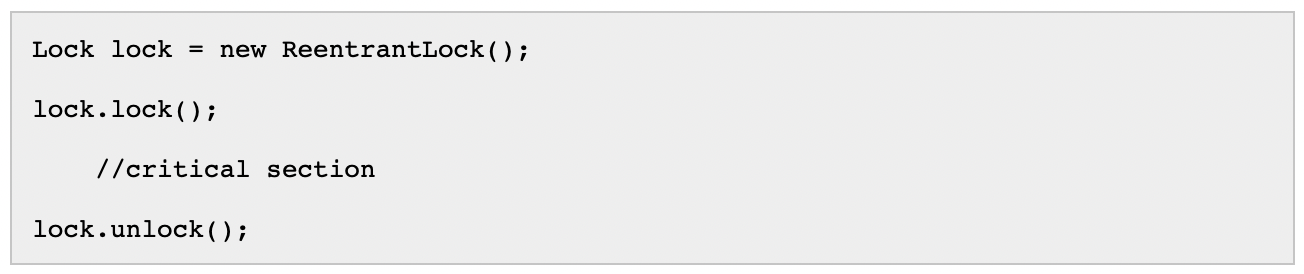
Since Java **Lock is an interface**, we **cannot create an instance of Lock directly**. We must create an instance of a **class that implements the Lock** interface e.g., **ReentrantLock**.

To create an instance of the ReentrantLock class we simply use the new operator, like this:



To **lock the Lock instance** we must call its **lock() method**. To **unlock the Lock instance** we must call its **unlock() method**.

Here is an example of locking and unlocking a Java lock instance:

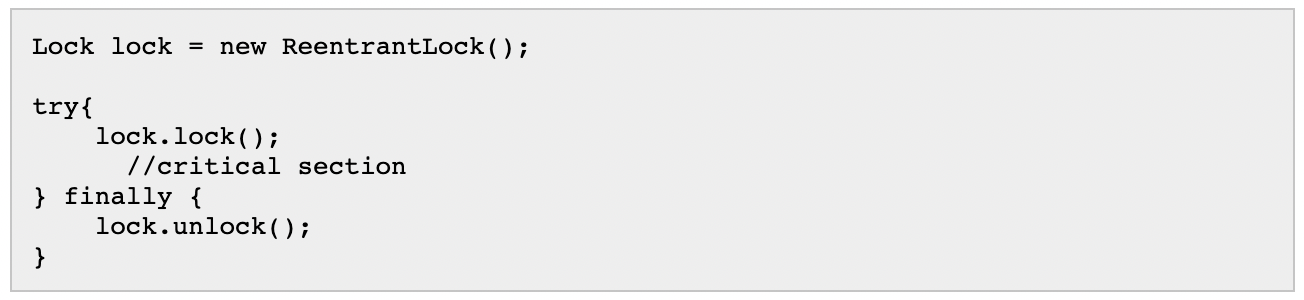


First, a Lock is created. Then its lock() method is called. Now the Lock instance is locked. Any other thread calling lock() will be blocked until the thread that locked the lock calls unlock(). Finally, unlock() is called, and the Lock is now unlocked so other threads can lock it.

**Fail-safe Lock and Unlock**

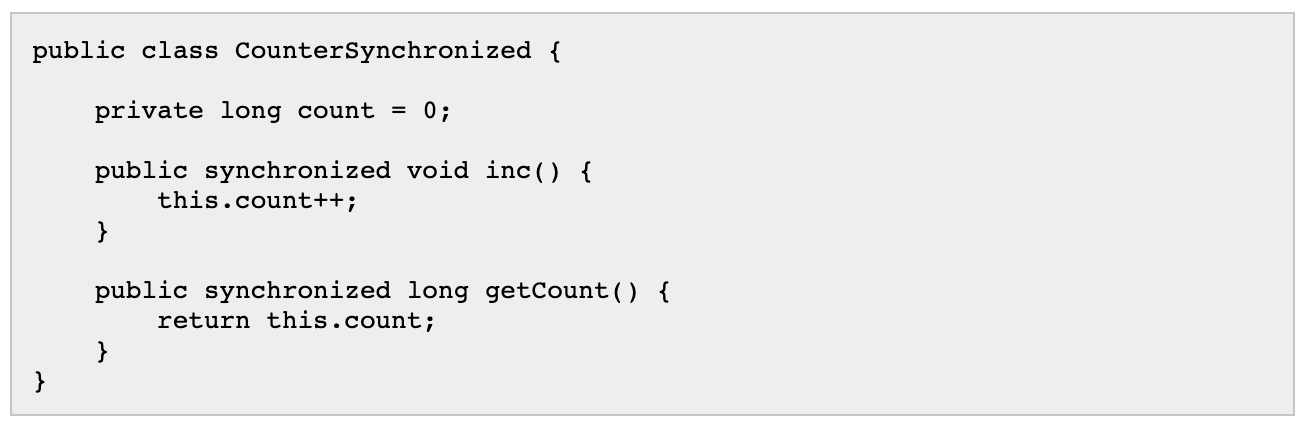
If we look at the example in the previous section, imagine **what happens if an exception is thrown between the call to lock.lock() and lock.unlock()**. The **exception would interrupt the program flow**, and the **call to lock.unlock() would never be executed**. The **Lock would thus remain locked forever**.

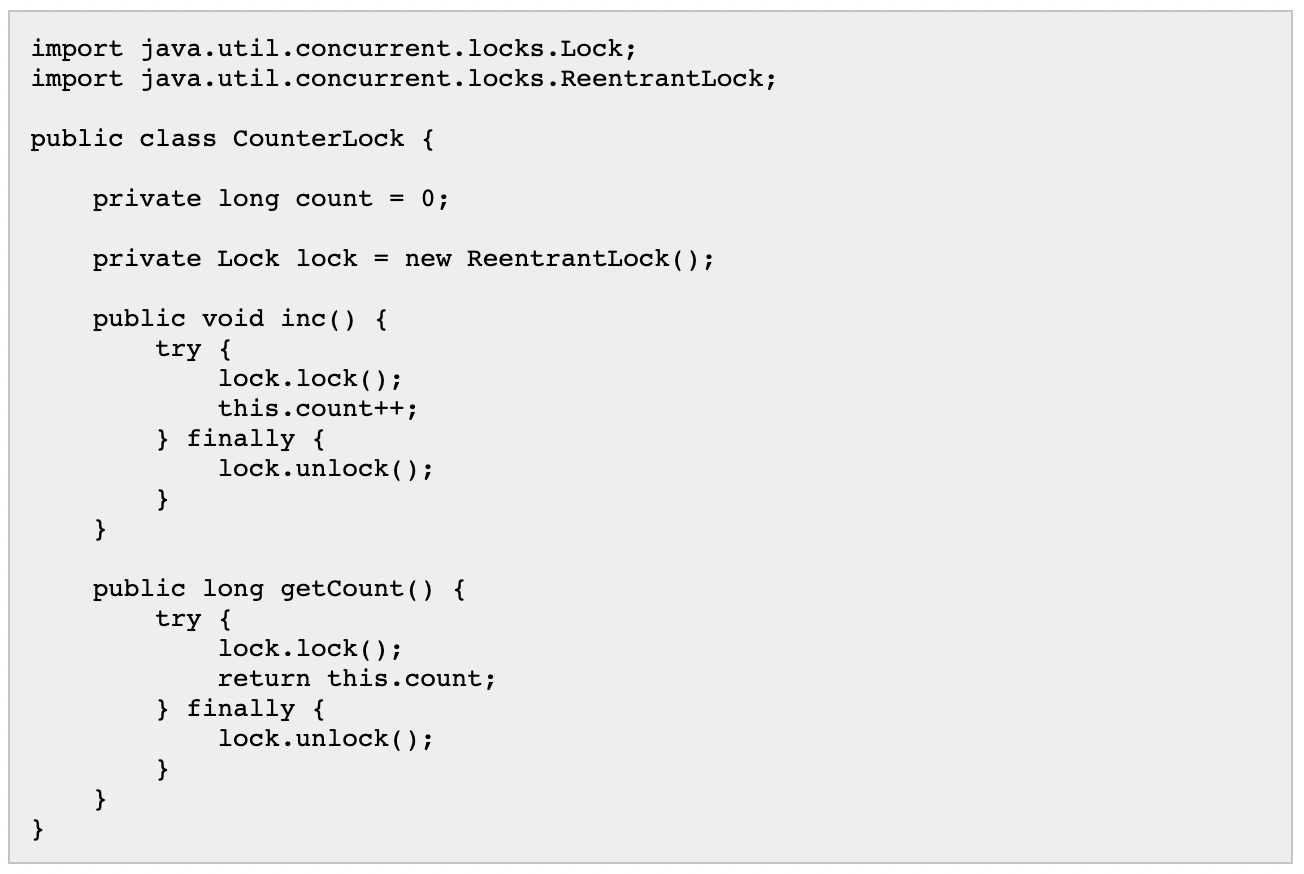
To avoid exceptions locking a Lock forever, we should lock and unlock it from within a try-finally block, like this:



To better understand how using a Lock looks different from using a synchronized block, we have created two simple concurrent Counter classes which protect their internal count in different ways.

The **first class uses a synchronized block**, and the **second class uses a Java Lock**:





**A lock is called reentrant** **if the thread that holds the lock** **can lock it again**. A **non-reentrant lock is a lock that cannot be locked again** if locked, **not even by the thread that holds the lock**. **Non-reentrant locks may result in** [**reentrance lockout**](http://tutorials.jenkov.com/java-concurrency/reentrance-lockout.html) which is a situation **similar to a** [**deadlock**](http://tutorials.jenkov.com/java-concurrency/deadlock.html).

The **ReentrantLock class is a reentrant lock**. That means, that even if a thread holds the lock it can lock it again. Consequently, the **thread must unlock it as many times as it has locked it,** in order **to fully unlock the Reentrant lock for other threads**.

A reentrant lock is useful in certain concurrent designs. **Below is a** **concurrent implementation of a calculator**. The calculator can hold the current result internally and offers a set of methods that can perform calculations on that result.



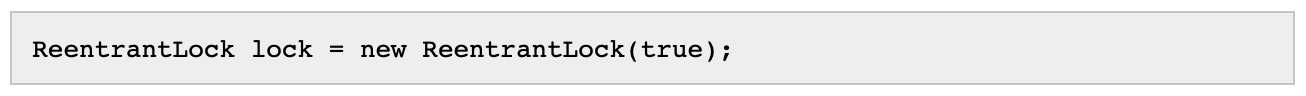
Notice how the **calculate() method** both **locks the Calculator instance's Lock** before performing any calculations, and **also call the add() and subtract() methods which also locks the lock**. Because theReentrantLock is reentrant, this does not cause any problems.

**Lock Fairness**

**An unfair lock does not guarantee the order in which threads waiting to lock the lock will be given access to lock it**. That means, that **a waiting thread could risk waiting forever**, if **other threads keep trying to lock the lock**, and are **given priority** over the waiting thread. This **situation** **can lead to** **starvation**.

The **ReentrantLock behaviour is unfair by default**. However, you can tell it to operate in fair mode via its constructor. The **ReentrantLock class has a constructor** **that takes a boolean parameter specifying whether the ReentrantLock should provide fairness or not to waiting threads**.

Here is an example of creating a ReentrantLock instance using fair mode:



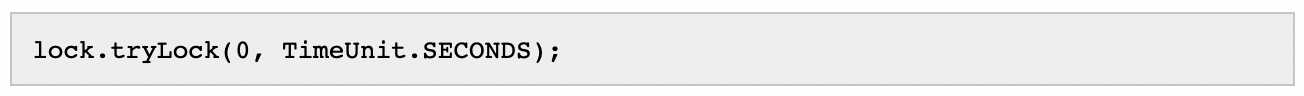
The **Java Lock interface** contains the following **primary methods**:

* **lock()**
  + The lock() method **locks the Lock instance if possible**. If the **Lock instance is already locked**, the **thread calling lock() is blocked until the Lock is unlocked**.
* **tryLock()**
  + The tryLock() method **attempts to lock the Lock instance immediately**. It **returns true if the locking succeeds**, **false if Lock is already locked**. This **method never blocks**.
* **tryLock(long timeout, TimeUnit timeUnit)**
  + The tryLock(long timeout, TimeUnit timeUnit) **works like the tryLock() method**, except **it** **waits up the given timeout before giving up trying to lock the Lock**.
* **unlock()**
  + The unlock() method **unlocks the Lock instance**. Typically, a Lock implementation will **only allow the thread that has locked the Lock to call this method**. **Other threads calling this method may result in an unchecked exception** (RuntimeException).

The example given in the javadocs is as follows:



Please note, that the **method tryLock() with no parameters** **does not respect the fairness mode of the ReentrantLock**. **To get fairness we must use the tryLock(long timeout, TimeUnit unit) method** **instead**, like this:



**Semaphore**