# What is the ThreadLocal class? How and why would you use it?

[The documentation](http://docs.oracle.com/javase/7/docs/api/java/lang/ThreadLocal.html) says it very well: "each thread that accesses [a thread-local variable] (via its get or set method) has its own, independently initialized copy of the variable".

You use one when each thread must have its own copy of something. By default, data is shared between threads.

This class provides thread local variable. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable.

* Basically it is an another way to achieve thread safety apart from writing immutable classes.
* Since Object is no more shared there is no requirement of Synchronization which can improve scalability and performance of application.
* It extends class Object.
* ThreadLocal provides thread restriction which is extension of local variable. ThreadLocal are visible only in single thread. No two thread can see each others thread local variable.
* ThreadLocal variable are generally private static field in classes and maintain its state inside thread.
* ThreadLocal in Java had been introduced on JDK 1.2 but was later generified in JDK 1.5 to introduce type safety on ThreadLocal variable.
* Each thread holds an exclusive copy of ThreadLocal variable which becomes eligible to Garbage collection after thread finished or died, normally or due to any Exception, Given those ThreadLocal variable doesn't have any other live references.
* One possible (and common) use is when you have some object that is not thread-safe, but you want to avoid [synchronizing](https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html) access to that object Instead, give each thread its own instance of the object.

**Constructor:**  
**ThreadLocal():** This creates a thread local variable.  
**Methods:**

1. **Object get():** This method returns the value in the current thread’s copy of this thread-local variable. If the variable has no value for the current thread, it is first initialized to the value returned by an invocation of the initialValue() method.
2. **Syntax:** public Object get().
3. **Returns:** the current thread's value of this thread-local.
4. **Exception:** NA.
5. **void set(Object value):** This method sets the current thread’s copy of this thread-local variable to the specified value. Most subclasses will have no need to override this method, relying solely on the initialValue() method to set the values of thread-locals.

**void remove():** This method removes the current thread’s value for this thread-local variable. If this thread-local variable is subsequently read by the current thread, its value will be reinitialized by invoking its initialValue() method, unless its value is set by the current thread in the interim. This may result in multiple invocations of the initialValue method in the current thread.

**Object initialValue():** This method returns the current thread’s initial value for this thread-local variable.

import java.lang.\*;

class NewThread extends Thread {

private static ThreadLocal gfg = new ThreadLocal(){

protected Object initialValue(){ return new Integer(question--); }

};

private static int question = 15;

NewThread(String name){

super(name);

start();

}

public void run(){

for (int i = 0; i < 2; i++)

System.out.println(getName() + " " + gfg.get());

}}

public class ThreadLocalDemo {

public static void main(String[] args) {

NewThread t1 = new NewThread("quiz1");

NewThread t2 = new NewThread("quiz2");

}

}

Output:

quiz2 14

quiz1 15

quiz1 15

quiz2 14

[**When and how should I use a ThreadLocal variable?**](https://stackoverflow.com/questions/817856/when-and-how-should-i-use-a-threadlocal-variable)

Many frameworks use ThreadLocals to maintain some context related to the current thread. For example when the current transaction is stored in a ThreadLocal, you don't need to pass it as a parameter through every method call, in case someone down the stack needs access to it.

Web applications might store information about the current request and session in a ThreadLocal, so that the application has easy access to them.

Two use cases where threadlocal variable can be used -  
1- When we have a requirement to associate state with a thread (e.g., a user ID or Transaction ID). That usually happens with a web application that every request going to a servlet has a unique transactionID associated with it.

// This class will provide a thread local variable which

// will provide a unique ID for each thread

class ThreadId {

// Atomic integer containing the next thread ID to be assigned

private static final AtomicInteger nextId = new AtomicInteger(0);

// Thread local variable containing each thread's ID

private static final ThreadLocal<Integer> threadId =

ThreadLocal.<Integer>withInitial(()-> {return nextId.getAndIncrement();});

// Returns the current thread's unique ID, assigning it if necessary

public static int get() {

return threadId.get();

}

}

2- Another use case is when we want to have a thread safe instance and we don't want to use synchronization as the performance cost with synchronization is more. One such case is when SimpleDateFormat is used. Since SimpleDateFormat is not thread safe so we have to provide mechanism to make it thread safe.

public class ThreadLocalDemo1 implements Runnable {

// threadlocal variable is created

private static final ThreadLocal<SimpleDateFormat> dateFormat = new ThreadLocal<SimpleDateFormat>(){

@Override

protected SimpleDateFormat initialValue(){

System.out.println("Initializing SimpleDateFormat for - " + Thread.currentThread().getName() );

return new SimpleDateFormat("dd/MM/yyyy");

}

};

public static void main(String[] args) {

ThreadLocalDemo1 td = new ThreadLocalDemo1();

// Two threads are created

Thread t1 = new Thread(td, "Thread-1");

Thread t2 = new Thread(td, "Thread-2");

t1.start();

t2.start();

}

@Override

public void run() {

System.out.println("Thread run execution started for " + Thread.currentThread().getName());

System.out.println("Date formatter pattern is " + dateFormat.get().toPattern());

System.out.println("Formatted date is " + dateFormat.get().format(new Date()));

}

}

# What is the difference between atomic / volatile / synchronized?

**volatile** is a keyword. volatile forces all threads to get latest value of the variable from main memory instead of cache. No locking is required to access volatile variables.All threads can access volatile variable value at same time.Using volatile variables reduces the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable.

This means that changes to a volatile variable are always visible to other threads. What's more, it also means that when a thread reads a volatile variable, it sees not just the latest change to the volatile, but also the side effects of the code that led up the change.

When to use: One thread modifies the data and other threads have to read latest value of data. Other threads will take some action but they won't update data.

**AtomicXXX**: AtomicXXX classes support lock-free thread-safe programming on single variables. These AtomicXXX classes (like AtomicInteger) resolves memory inconsistency errors / side effects of modification of volatile variables, which have been accessed in multiple threads.When to use: Multiple threads can read and modify data.

**synchronized**: synchronized is keyword used to guard a method or code block. By making method as synchronized has two effects:

First, it is not possible for two invocations of synchronized methods on the same object to interleave.

When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

Second, when a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

When to use: Multiple threads can read and modify data. Your business logic not only update the data but also executes atomic operations

AtomicXXX is equivalent of volatile + synchronized even though the implementation is different.

AmtomicXXX extends volatile variables + compareAndSet methods but does not use synchronization.If you have only one thread modifying your boolean, you can use a volatile boolean (usually you do this to define a stop variable checked in the thread's main loop).However, if you have multiple threads modifying the boolean, you should use an AtomicBoolean.

# Compare the sleep() and wait() methods in Java, including when and why you would use one vs. the other.

**sleep**() is a blocking operation that keeps a hold on the monitor / lock of the shared object for the specified number of milliseconds.

**wait**(), on the other hand, simply pauses the thread until either (a) the specified number of milliseconds have elapsed or (b) it receives a desired notification from another thread (whichever is first), without keeping a hold on the monitor/lock of the shared object.

sleep() is most commonly used for polling, or to check for certain results, at a regular interval.  
wait() is generally used in multithreaded applications, in conjunction with notify() / notifyAll(), to achieve synchronization and avoid race conditions.

# How can you catch an exception thrown by another thread in Java?

This can be done using Thread.UncaughtExceptionHandler.

// create our uncaught exception handler

Thread.UncaughtExceptionHandler handler = new Thread.UncaughtExceptionHandler() {

public void uncaughtException(Thread th, Throwable ex) {

System.out.println("Uncaught exception: " + ex);

}

};

// create another thread

Thread otherThread = new Thread() {

public void run() {

System.out.println("Sleeping ...");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

System.out.println("Interrupted.");

}

System.out.println("Throwing exception ...");

throw new RuntimeException();

}

};

// set our uncaught exception handler as the one to be used when the new thread// throws an uncaught exception

otherThread.setUncaughtExceptionHandler(handler);

// start the other thread - our uncaught exception handler will be invoked when// the other thread throws an uncaught exception

otherThread.start();

# Why do you need to use synchronized methods or blocks

**It's not a matter of better, just different.**

When you synchronize a method, you are effectively synchronizing to the object itself. In the case of a static method, you're synchronizing to the class of the object. So the following two pieces of code execute the same way:

public synchronized int getCount() {

// ...

}

This is just like you wrote this.

public int getCount() {

synchronized (this) {

// ...

}

}

If you want to control synchronization to a specific object, or you only want *part* of a method to be synchronized to the object, then specify a synchronized block. If you use the synchronized keyword on the method declaration, it will synchronize the whole method to the object or class.

Difference between **synchronized block** and **synchronized method** are following:

1. synchronized block reduce scope of lock, **but** synchronized method's scope of lock is whole method.
2. synchronized block has better performance as only the critical section is locked **but** synchronized method has poor performance than block.
3. synchronized block provide granular control over lock **but** synchronized method lock either on current object represented by this or class level lock.
4. synchronized block can throw NullPointerException **but** synchronized method doesn't throw.
5. **synchronized block:** synchronized(this){}

**synchronized method:** public synchronized void fun(){}

# **Describe the Different States of a Thread and When Do the State Transitions Occur.**

The state of a Thread can be checked using the Thread.getState() method. Different states of a Thread are described in the Thread.State enum. They are:

* **NEW** — a new Thread instance that was not yet started via Thread.start()
* **RUNNABLE** — a running thread. It is called runnable because at any given time it could be either running or waiting for the next quantum of time from the thread scheduler. A NEW thread enters the RUNNABLE state when you call Thread.start() on it
* **BLOCKED** — a running thread becomes blocked if it needs to enter a synchronized section but cannot do that due to another thread holding the monitor of this section
* **WAITING** — a thread enters this state if it waits for another thread to perform a particular action. For instance, a thread enters this state upon calling the Object.wait() method on a monitor it holds, or the Thread.join() method on another thread
* **TIMED\_WAITING** — same as the above, but a thread enters this state after calling timed versions of Thread.sleep(), Object.wait(), Thread.join() and some other methods
* **TERMINATED** — a thread has completed the execution of its Runnable.run() method and terminated

# **What Is the Difference Between the Runnable and Callable Interfaces? How Are They Used?**

The Runnable interface has a single run method. It represents a unit of computation that has to be run in a separate thread. The Runnable interface does not allow this method to return value or to throw unchecked exceptions.

The Callable interface has a single call method and represents a task that has a value. That's why the call method returns a value. It can also throw exceptions. Callable is generally used in ExecutorService instances to start an asynchronous task and then call the returned Future instance to get its value.

# **What Is a Daemon Thread, What Are Its Use Cases? How Can You Create a Daemon Thread?**

A daemon thread is a thread that does not prevent JVM from exiting. When all non-daemon threads are terminated, the JVM simply abandons all remaining daemon threads. Daemon threads are usually used to carry out some supportive or service tasks for other threads, but you should take into account that they may be abandoned at any time.

To start a thread as a daemon, you should use the setDaemon() method before calling start():

Thread daemon = **new** Thread(()

-> System.out.println("Hello from daemon!"));

daemon.setDaemon(**true**);

daemon.start();

Curiously, if you run this as a part of the main() method, the message might not get printed. This could happen if the main() thread would terminate before the daemon would get to the point of printing the message. You generally should not do any I/O in daemon threads, as they won't even be able to execute their finally blocks and close the resources if abandoned.

# **What Is the Thread’s Interrupt Flag? How Can You Set and Check It? How Does It Relate to the Interruptedexception?**

The interrupt flag, or interrupt status, is an internal Thread flag that is set when the thread is interrupted. To set it, simply call thread.interrupt() on the thread object.

If a thread is currently inside one of the methods that throw InterruptedException (wait, join, sleep etc.), then this method immediately throws InterruptedException. The thread is free to process this exception according to its own logic.

If a thread is not inside such method and thread.interrupt() is called, nothing special happens. It is thread's responsibility to periodically check the interrupt status using static Thread.interrupted() or instance isInterrupted() method. The difference between these methods is that the static Thread.interrupted() clears the interrupt flag, while isInterrupted() does not.

# **If Two Threads Call a Synchronized Method on Different Object Instances Simultaneously, Could One of These Threads Block? What If the Method Is Static?**

If the method is an instance method, then the instance acts as a monitor for the method. Two threads calling the method on different instances acquire different monitors, so none of them gets blocked.

If the method is static, then the monitor is the Class object. For both threads, the monitor is the same, so one of them will probably block and wait for another to exit the synchronized method.

# **What Is the Purpose of the Wait, Notify and Notifyall Methods of the Object Class?**

A thread that owns the object's monitor (for instance, a thread that has entered a synchronized section guarded by the object) may call object.wait() to temporarily release the monitor and give other threads a chance to acquire the monitor. This may be done, for instance, to wait for a certain condition.

When another thread that acquired the monitor fulfills the condition, it may call object.notify() or object.notifyAll() and release the monitor. The notify method awakes a single thread in the waiting state, and the notifyAll method awakes all threads that wait for this monitor, and they all compete for re-acquiring the lock.

The following BlockingQueue implementation shows how multiple threads work together via the wait-notify pattern. If we put an element into an empty queue, all threads that were waiting in the take method wake up and try to receive the value. If we put an element into a full queue, the put method waits for the call to the get method. The get method removes an element and notifies the threads waiting in the put method that the queue has an empty place for a new item.

**public** **class** **BlockingQueue**<**T**> {

**private** List<T> queue = **new** LinkedList<T>();

**private** **int** limit = 10;

**public** **synchronized** **void** **put**(T item) {

**while** (queue.size() == limit) {

**try** {

wait();

} **catch** (InterruptedException e) {}

}

**if** (queue.isEmpty()) {

notifyAll();

}

queue.add(item);

}

**public** **synchronized** T **take**() **throws** InterruptedException {

**while** (queue.isEmpty()) {

**try** {

wait();

} **catch** (InterruptedException e) {}

}

**if** (queue.size() == limit) {

notifyAll();

}

**return** queue.remove(0);

}

}

# **Describe the Conditions of Deadlock, Livelock, and Starvation. Describe the Possible Causes of These Conditions.**

**Deadlock** is a condition within a group of threads that cannot make progress because every thread in the group has to acquire some resource that is already acquired by another thread in the group. The most simple case is when two threads need to lock both of two resources to progress, the first resource is already locked by one thread, and the second by another. These threads will never acquire a lock to both resources and thus will never progress.

**Livelock** is a case of multiple threads reacting to conditions, or events, generated by themselves. An event occurs in one thread and has to be processed by another thread. During this processing, a new event occurs which has to be processed in the first thread, and so on. Such threads are alive and not blocked, but still, do not make any progress because they overwhelm each other with useless work.

**Starvation** is a case of a thread unable to acquire resource because other thread (or threads) occupy it for too long or have higher priority. A thread cannot make progress and thus is unable to fulfill useful work.

# **What Are the Available Implementations of Executorservice in the Standard Library?**

The ExecutorService interface has three standard implementations:

* **ThreadPoolExecutor** — for executing tasks using a pool of threads. Once a thread is finished executing the task, it goes back into the pool. If all threads in the pool are busy, then the task has to wait for its turn.
* **ScheduledThreadPoolExecutor** allows to schedule task execution instead of running it immediately when a thread is available. It can also schedule tasks with fixed rate or fixed delay.
* **ForkJoinPool** is a special ExecutorService for dealing with recursive algorithms tasks. If you use a regular ThreadPoolExecutor for a recursive algorithm, you will quickly find all your threads are busy waiting for the lower levels of recursion to finish. The ForkJoinPool implements the so-called work-stealing algorithm that allows it to use available threads more efficiently.

# **ExecutorService**

[*ExecutorService*](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ExecutorService.html) is a JDK API that simplifies running tasks in asynchronous mode. Generally speaking, ExecutorService automatically provides a pool of threads and an API for assigning tasks to it.

## **2. Instantiating** ExecutorService

### **2.1. Factory Methods of the** Executors **Class**

The easiest way to create ExecutorService is to use one of the factory methods of the Executors class.

For example, the following line of code will create a thread pool with 10 threads:

ExecutorService executor = Executors.newFixedThreadPool(10);

There are several other factory methods to create a predefined ExecutorService that meets specific use cases. To find the best method for your needs, consult [Oracle's official documentation](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Executors.html).

### **2.2. Directly Create an** ExecutorService

Because ExecutorService is an interface, an instance of any its implementations can be used. There are several implementations to choose from in the [*java.util.concurrent*](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Executors.html) package, or you can create your own.

For example, the ThreadPoolExecutor class has a few constructors that we can use to configure an executor service and its internal pool:

ExecutorService executorService =

**new** ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,

**new** LinkedBlockingQueue<Runnable>());

You may notice that the code above is very similar to the [source code](https://github.com/openjdk-mirror/jdk7u-jdk/blob/master/src/share/classes/java/util/concurrent/Executors.java#L133) of the factory method newSingleThreadExecutor(). For most cases, a detailed manual configuration isn't necessary.

## **3. Assigning Tasks to the** ExecutorService

ExecutorService can execute Runnable and Callable tasks. To keep things simple in this article, two primitive tasks will be used. Notice that we use lambda expressions here instead of anonymous inner classes:

Runnable runnableTask = () -> {

**try** {

TimeUnit.MILLISECONDS.sleep(300);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

};

Callable<String> callableTask = () -> {

TimeUnit.MILLISECONDS.sleep(300);

**return** "Task's execution";

};

List<Callable<String>> callableTasks = **new** ArrayList<>();

callableTasks.add(callableTask);

callableTasks.add(callableTask);

callableTasks.add(callableTask);

We can assign tasks to the ExecutorService using several methods including execute(), which is inherited from the Executor interface, and also submit(), invokeAny() and invokeAll().

The **execute()** method is void and doesn't give any possibility to get the result of a task's execution or to check the task's status (is it running):

executorService.execute(runnableTask);

**submit()** submits a Callable or a Runnable task to an ExecutorService and returns a result of type Future:

Future<String> future =

executorService.submit(callableTask);

**invokeAny()** assigns a collection of tasks to an ExecutorService, causing each to run, and returns the result of a successful execution of one task (if there was a successful execution):

String result = executorService.invokeAny(callableTasks);

***invokeAll()*** assigns a collection of tasks to an ExecutorService, causing each to run, and returns the result of all task executions in the form of a list of objects of type Future:

List<Future<String>> futures = executorService.invokeAll(callableTasks);

Before going further, we need to discuss two more items: shutting down an ExecutorService and dealing with Future return types.

## **4. Shutting Down an** ExecutorService

In general, the ExecutorService will not be automatically destroyed when there is no task to process. It will stay alive and wait for new work to do.

In some cases this is very helpful, such as when an app needs to process tasks that appear on an irregular basis or the task quantity is not known at compile time.

On the other hand, an app could reach its end but not be stopped because a waiting ExecutorService will cause the JVM to keep running.

To properly shut down an ExecutorService, we have the shutdown() and shutdownNow() APIs.

The ***shutdown()*** method doesn't cause immediate destruction of the ExecutorService. It will make the ExecutorService stop accepting new tasks and shut down after all running threads finish their current work:

executorService.shutdown();

The **shutdownNow()** method tries to destroy the ExecutorService immediately, but it doesn't guarantee that all the running threads will be stopped at the same time:

List<Runnable> notExecutedTasks = executorService.shutDownNow();

This method returns a list of tasks that are waiting to be processed. It is up to the developer to decide what to do with these tasks.

One good way to shut down the ExecutorService (which is also [recommended by Oracle](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ExecutorService.html)) is to use both of these methods combined with the **awaitTermination()** method:

executorService.shutdown();

**try** {

**if** (!executorService.awaitTermination(800, TimeUnit.MILLISECONDS)) {

executorService.shutdownNow();

}

} **catch** (InterruptedException e) {

executorService.shutdownNow();

}

With this approach, the ExecutorService will first stop taking new tasks and then wait up to a specified period of time for all tasks to be completed. If that time expires, the execution is stopped immediately.

## **The** Future **Interface**

The submit() and invokeAll() methods return an object or a collection of objects of type Future, which allows us to get the result of a task's execution or to check the task's status (is it running).

The Future interface provides a special blocking method get(), which returns an actual result of the Callable task's execution or null in the case of a Runnable task:

Future<String> future = executorService.submit(callableTask);

String result = **null**;

**try** {

result = future.get();

} **catch** (InterruptedException | ExecutionException e) {

e.printStackTrace();

}

Calling the get() method while the task is still running will cause execution to block until the task properly executes and the result is available.

With very long blocking caused by the get() method, an application's performance can degrade. If the resulting data is not crucial, it is possible to avoid such a problem by using timeouts:

String result = future.get(200, TimeUnit.MILLISECONDS);

If the execution period is longer than specified (in this case, 200 milliseconds), a TimeoutException will be thrown.

We can use the isDone() method to check if the assigned task already processed or not.

The Future interface also provides for canceling task execution with the cancel() method and checking the cancellation with the isCancelled() method:

**boolean** canceled = future.cancel(**true**);

**boolean** isCancelled = future.isCancelled();

## **6. The** ScheduledExecutorService **Interface**

The ScheduledExecutorService runs tasks after some predefined delay and/or periodically.

Once again, the best way to instantiate a ScheduledExecutorService is to use the factory methods of the Executors class.

For this section, we use a ScheduledExecutorService with one thread:

ScheduledExecutorService executorService = Executors

.newSingleThreadScheduledExecutor();

To schedule a single task's execution after a fixed delay, use the scheduled() method of the ScheduledExecutorService.

Two scheduled() methods allow you to execute Runnable or Callable tasks:

Future<String> resultFuture =

executorService.schedule(callableTask, 1, TimeUnit.SECONDS);

The scheduleAtFixedRate() method lets us run a task periodically after a fixed delay. The code above delays for one second before executing callableTask.

The following block of code will run a task after an initial delay of 100 milliseconds. And after that, it will run the same task every 450 milliseconds:

Future<String> resultFuture = service

.scheduleAtFixedRate(runnableTask, 100, 450, TimeUnit.MILLISECONDS);

If the processor needs more time to run an assigned task than the period parameter of the scheduleAtFixedRate() method, the ScheduledExecutorService will wait until the current task is completed before starting the next.

If it is necessary to have a fixed length delay between iterations of the task, scheduleWithFixedDelay() should be used.

For example, the following code will guarantee a 150-millisecond pause between the end of the current execution and the start of another one:

service.scheduleWithFixedDelay(task, 100, 150, TimeUnit.MILLISECONDS);

According to the scheduleAtFixedRate() and scheduleWithFixedDelay() method contracts, period execution of the task will end at the termination of the ExecutorService or if an exception is thrown during task execution.

## **7.** ExecutorService **vs Fork/Join**

After the release of Java 7, many developers decided to replace the ExecutorService framework with the fork/join framework.

This is not always the right decision, however. Despite the simplicity and frequent performance gains associated with fork/join, it reduces developer control over concurrent execution.

ExecutorService gives the developer the ability to control the number of generated threads and the granularity of tasks that should be run by separate threads. The best use case for ExecutorService is the processing of independent tasks, such as transactions or requests according to the scheme “one thread for one task.”

## **Conclusion**

Despite the relative simplicity of ExecutorService, there are a few common pitfalls.

Let's summarize them:

**Keeping an unused ExecutorService alive**: See the detailed explanation in Section 4 on how to shut down an ExecutorService.

**Wrong thread-pool capacity while using fixed length thread pool**: It is very important to determine how many threads the application will need to run tasks efficiently. A too-large thread pool will cause unnecessary overhead just to create threads that will mostly be in the waiting mode. Too few can make an application seem unresponsive because of long waiting periods for tasks in the queue.

**Calling a Future‘s get() method after task cancellation**: Attempting to get the result of an already canceled task triggers a CancellationException.

**Unexpectedly long blocking with Future‘s get() method**: We should use timeouts to avoid unexpected waits.

# **CountDownLatch**

## **Introduction**

In this article, we'll give a guide to the *[CountDownLatch](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/CountDownLatch.html)* class and demonstrate how it can be used in a few practical examples.

Essentially, by using a CountDownLatch we can cause a thread to block until other threads have completed a given task.

## **Usage in Concurrent Programming**

Simply put, a CountDownLatch has a counter field, which you can decrement as we require. We can then use it to block a calling thread until it's been counted down to zero.

If we were doing some parallel processing, we could instantiate the CountDownLatch with the same value for the counter as a number of threads we want to work across. Then, we could just call countdown() after each thread finishes, guaranteeing that a dependent thread calling await() will block until the worker threads are finished.

## **Waiting for a Pool of Threads to Complete**

Let's try out this pattern by creating a Worker and using a CountDownLatch field to signal when it has completed:

**public** **class** **Worker** **implements** **Runnable** {

**private** List<String> outputScraper;

**private** CountDownLatch countDownLatch;

**public** **Worker**(List<String> outputScraper, CountDownLatch countDownLatch) {

**this**.outputScraper = outputScraper;

**this**.countDownLatch = countDownLatch;

}

@Override

**public** **void** **run**() {

doSomeWork();

outputScraper.add("Counted down");

countDownLatch.countDown();

}

}

Then, let's create a test in order to prove that we can get a CountDownLatch to wait for the Worker instances to complete:

@Test

**public** **void** **whenParallelProcessing\_thenMainThreadWillBlockUntilCompletion**()

**throws** InterruptedException {

List<String> outputScraper = Collections.synchronizedList(**new** ArrayList<>());

CountDownLatch countDownLatch = **new** CountDownLatch(5);

List<Thread> workers = Stream

.generate(() -> **new** Thread(**new** Worker(outputScraper, countDownLatch)))

.limit(5)

.collect(toList());

workers.forEach(Thread::start);

countDownLatch.await();

outputScraper.add("Latch released");

assertThat(outputScraper)

.containsExactly(

"Counted down",

"Counted down",

"Counted down",

"Counted down",

"Counted down",

"Latch released"

);

}

Naturally “Latch released” will always be the last output – as it's dependant on the CountDownLatch releasing.

Note that if we didn't call await(), we wouldn't be able to guarantee the ordering of the execution of the threads, so the test would randomly fail.

## 4. **A Pool of Threads Waiting to Begin**

If we took the previous example, but this time started thousands of threads instead of five, it's likely that many of the earlier ones will have finished processing before we have even called start() on the later ones. This could make it difficult to try and reproduce a concurrency problem, as we wouldn't be able to get all our threads to run in parallel.

To get around this, let's get the CountdownLatch to work differently than in the previous example. Instead of blocking a parent thread until some child threads have finished, we can block each child thread until all the others have started.

Let's modify our run() method so it blocks before processing:

**public** **class** **WaitingWorker** **implements** **Runnable** {

**private** List<String> outputScraper;

**private** CountDownLatch readyThreadCounter;

**private** CountDownLatch callingThreadBlocker;

**private** CountDownLatch completedThreadCounter;

**public** **WaitingWorker**(

List<String> outputScraper,

CountDownLatch readyThreadCounter,

CountDownLatch callingThreadBlocker,

CountDownLatch completedThreadCounter) {

**this**.outputScraper = outputScraper;

**this**.readyThreadCounter = readyThreadCounter;

**this**.callingThreadBlocker = callingThreadBlocker;

**this**.completedThreadCounter = completedThreadCounter;

}

@Override

**public** **void** **run**() {

readyThreadCounter.countDown();

**try** {

callingThreadBlocker.await();

doSomeWork();

outputScraper.add("Counted down");

} **catch** (InterruptedException e) {

e.printStackTrace();

} **finally** {

completedThreadCounter.countDown();

}

}

}

Now, let's modify our test so it blocks until all the Workers have started, unblocks the Workers, and then blocks until the Workers have finished:

@Test

**public** **void** **whenDoingLotsOfThreadsInParallel\_thenStartThemAtTheSameTime**()

**throws** InterruptedException {

List<String> outputScraper = Collections.synchronizedList(**new** ArrayList<>());

CountDownLatch readyThreadCounter = **new** CountDownLatch(5);

CountDownLatch callingThreadBlocker = **new** CountDownLatch(1);

CountDownLatch completedThreadCounter = **new** CountDownLatch(5);

List<Thread> workers = Stream

.generate(() -> **new** Thread(**new** WaitingWorker(

outputScraper, readyThreadCounter, callingThreadBlocker, completedThreadCounter)))

.limit(5)

.collect(toList());

workers.forEach(Thread::start);

readyThreadCounter.await();

outputScraper.add("Workers ready");

callingThreadBlocker.countDown();

completedThreadCounter.await();

outputScraper.add("Workers complete");

assertThat(outputScraper)

.containsExactly(

"Workers ready",

"Counted down",

"Counted down",

"Counted down",

"Counted down",

"Counted down",

"Workers complete"

);

}

This pattern is really useful for trying to reproduce concurrency bugs, as can be used to force thousands of threads to try and perform some logic in parallel.

## **5. Terminating a**CountdownLatch**Early**

Sometimes, we may run into a situation where the Workers terminate in error before counting down the CountDownLatch. This could result in it never reaching zero and await() never terminating:

@Override

**public** **void** **run**() {

**if** (**true**) {

**throw** **new** RuntimeException("Oh dear, I'm a BrokenWorker");

}

countDownLatch.countDown();

outputScraper.add("Counted down");

}

Let's modify our earlier test to use a BrokenWorker, in order to show how await() will block forever:

## **6. Conclusion**

In this quick guide, we've demonstrated how we can use a CountDownLatch in order to block a thread until other threads have finished some processing.

We've also shown how it can be used to help debug concurrency issues by making sure threads run in parallel.

# CyclicBarrier

CyclicBarrier works almost the same as CountDownLatch except that we can reuse it. Unlike CountDownLatch, it allows multiple threads to wait for each other using await() method(known as barrier condition) before invoking the final task.

We need to create a Runnable task instance to initiate the barrier condition:

**public** **class** **Task** **implements** **Runnable** {

**private** CyclicBarrier barrier;

**public** **Task**(CyclicBarrier barrier) {

**this**.barrier = barrier;

}

@Override

**public** **void** **run**() {

**try** {

LOG.info(Thread.currentThread().getName() +

" is waiting");

barrier.await();

LOG.info(Thread.currentThread().getName() +

" is released");

} **catch** (InterruptedException | BrokenBarrierException e) {

e.printStackTrace();

}

}

}

Now we can invoke some threads to race for the barrier condition:

**public** **void** **start**() {

CyclicBarrier cyclicBarrier = **new** CyclicBarrier(3, () -> {

// ...

LOG.info("All previous tasks are completed");

});

Thread t1 = **new** Thread(**new** Task(cyclicBarrier), "T1");

Thread t2 = **new** Thread(**new** Task(cyclicBarrier), "T2");

Thread t3 = **new** Thread(**new** Task(cyclicBarrier), "T3");

**if** (!cyclicBarrier.isBroken()) {

t1.start();

t2.start();

t3.start();

}

}

Here, the isBroken() method checks if any of the threads got interrupted during the execution time. We should always perform this check before performing the actual process.

# Semaphore

The Semaphore is used for blocking thread level access to some part of the physical or logical resource. A [semaphore](https://www.baeldung.com/cs/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 implement a semaphore:

**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()) {

**try** {

// ...

}

**finally** {

semaphore.release();

}

}

}

We can implement a Mutex like data-structure using Semaphore. More details on this[can be found here.](https://www.baeldung.com/java-semaphore)

# 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:

**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:

BaeldungThreadFactory factory = **new** BaeldungThreadFactory(

"BaeldungThreadFactory");

**for** (**int** i = 0; i < 10; i++) {

Thread t = factory.newThread(**new** Task());

t.start();

}

# **java.util.concurrent.Locks**

## **Overview**

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

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 synchronized 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:

Lock lock = ...;

lock.lock();

**try** {

// access to the shared resource

} **finally** {

lock.unlock();

}

In addition to the Lock interface*,*we have a ReadWriteLock interface which maintains a pair of locks, one for read-only operations, and one for the write operation. The read lock may be simultaneously held by multiple threads as long as there is no write.

ReadWriteLock declares methods to acquire read or write locks:

* ***Lock readLock()*** – returns the lock that's used for reading
* **Lock writeLock()** – returns the lock that's used for writing

## **4. Lock Implementations**

### **4.1.**ReentrantLock

ReentrantLock class implements the Lock interface. It offers the same concurrency and memory semantics, as the implicit monitor lock accessed using synchronized methods and statements, with extended capabilities.

Let's see, how we can use ReenrtantLock for synchronization:

**public** **class** **SharedObject** {

//...

ReentrantLock lock = **new** ReentrantLock();

**int** counter = 0;

**public** **void** **perform**() {

lock.lock();

**try** {

// Critical section here

count++;

} **finally** {

lock.unlock();

}

}

//...

}

We need to make sure that we are wrapping the lock() and the unlock() calls in the try-finally block to avoid the deadlock situations.

Let's see how the tryLock() works:

**public** **void** **performTryLock**(){

//...

**boolean** isLockAcquired = lock.tryLock(1, TimeUnit.SECONDS);

**if**(isLockAcquired) {

**try** {

//Critical section here

} **finally** {

lock.unlock();

}

}

//...

}

In this case, the thread calling tryLock(), will wait for one second and will give up waiting if the lock isn't available.

### **4.2.**ReentrantReadWriteLock

ReentrantReadWriteLock class implements the ReadWriteLock interface.

Let's see rules for acquiring the ReadLock or WriteLock by a thread:

* **Read Lock** – if no thread acquired the write lock or requested for it then multiple threads can acquire the read lock
* **Write Lock** – if no threads are reading or writing then only one thread can acquire the write lock

Let's see how to make use of the ReadWriteLock:

**public** **class** **SynchronizedHashMapWithReadWriteLock** {

Map<String,String> syncHashMap = **new** HashMap<>();

ReadWriteLock lock = **new** ReentrantReadWriteLock();

// ...

Lock writeLock = lock.writeLock();

**public** **void** **put**(String key, String value) {

**try** {

writeLock.lock();

syncHashMap.put(key, value);

} **finally** {

writeLock.unlock();

}

}

...

**public** String **remove**(String key){

**try** {

writeLock.lock();

**return** syncHashMap.remove(key);

} **finally** {

writeLock.unlock();

}

}

//...

}

For both the write methods, we need to surround the critical section with the write lock, only one thread can get access to it:

Lock readLock = lock.readLock();

//...

**public** String **get**(String key){

**try** {

readLock.lock();

**return** syncHashMap.get(key);

} **finally** {

readLock.unlock();

}

}

**public** **boolean** **containsKey**(String key) {

**try** {

readLock.lock();

**return** syncHashMap.containsKey(key);

} **finally** {

readLock.unlock();

}

}

For both read methods, we need to surround the critical section with the read lock. Multiple threads can get access to this section if no write operation is in progress.

### **4.3.**StampedLock

StampedLock is introduced in Java 8. It also supports both read and write locks. However, lock acquisition methods return a stamp that is used to release a lock or to check if the lock is still valid:

**public** **class** **StampedLockDemo** {

Map<String,String> map = **new** HashMap<>();

**private** StampedLock lock = **new** StampedLock();

**public** **void** **put**(String key, String value){

**long** stamp = lock.writeLock();

**try** {

map.put(key, value);

} **finally** {

lock.unlockWrite(stamp);

}

}

**public** String **get**(String key) **throws** InterruptedException {

**long** stamp = lock.readLock();

**try** {

**return** map.get(key);

} **finally** {

lock.unlockRead(stamp);

}

}

}

Another feature provided by StampedLock is optimistic locking. Most of the time read operations don't need to wait for write operation completion and as a result of this, the full-fledged read lock isn't required.

Instead, we can upgrade to read lock:

**public** String **readWithOptimisticLock**(String key) {

**long** stamp = lock.tryOptimisticRead();

String value = map.get(key);

**if**(!lock.validate(stamp)) {

stamp = lock.readLock();

**try** {

**return** map.get(key);

} **finally** {

lock.unlock(stamp);

}

}

**return** value;

}

## **5. Working With**Conditions

The Condition class provides the ability for a thread to wait for some condition to occur while executing the critical section.

This can occur when a thread acquires the access to the critical section but doesn't have the necessary condition to perform its operation. For example, a reader thread can get access to the lock of a shared queue, which still doesn't have any data to consume.

Traditionally Java provides wait(), notify() and notifyAll() methods for thread intercommunication. Conditions have similar mechanisms, but in addition, we can specify multiple conditions:

**public** **class** **ReentrantLockWithCondition** {

Stack<String> stack = **new** Stack<>();

**int** CAPACITY = 5;

ReentrantLock lock = **new** ReentrantLock();

Condition stackEmptyCondition = lock.newCondition();

Condition stackFullCondition = lock.newCondition();

**public** **void** **pushToStack**(String item){

**try** {

lock.lock();

**while**(stack.size() == CAPACITY) {

stackFullCondition.await();

}

stack.push(item);

stackEmptyCondition.signalAll();

} **finally** {

lock.unlock();

}

}

**public** String **popFromStack**() {

**try** {

lock.lock();

**while**(stack.size() == 0) {

stackEmptyCondition.await();

}

**return** stack.pop();

} **finally** {

stackFullCondition.signalAll();

lock.unlock();

}

}

}

## **6. Conclusion**

In this article, we have seen different implementations of the Lock interface and the newly introduced StampedLock class. We also explored how we can make use of the Condition class to work with multiple conditions.

# Custom Thread pool implementation in Java.

Thread pool executor requires a Queue for holding tasks and a collection of Worker Threads that will pick up tasks from the work queue start running them. Let us try to write our own simple Thread Pool Executor implementation.

It is a typical Producer Consumer Problem statement.

CustomThreadPoolExecutor Basic Implementation

**import java.util.concurrent.BlockingQueue;**

**import java.util.concurrent.LinkedBlockingQueue;**

**public class CustomThreadPoolExecutor {**

**private final BlockingQueue<Runnable> workerQueue;**

**private final Thread[] workerThreads;**

**public CustomThreadPoolExecutor(int numThreads) {**

**workerQueue = new LinkedBlockingQueue<>();**

**workerThreads = new Thread[numThreads];**

**int i = 0;**

**for (Thread t : workerThreads) {**

**t = new Worker("Custom Pool Thread " + ++i);**

**t.start();**

**}**

**}**

**public void addTask(Runnable r) {**

**try {**

**workerQueue.put(r);**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**class Worker extends Thread {**

**public Worker(String name) {**

**super(name);**

**}**

**public void run() {**

**while (true) {**

**try {**

**workerQueue.take().run();**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**} catch (Exception e) {**

**e.printStackTrace();**

**}**

**}**

**}**

**}**

**public static void main(String[] args) {**

**CustomThreadPoolExecutor threadPoolExecutor = new CustomThreadPoolExecutor(10);**

**threadPoolExecutor.addTask(() -> System.out.println("First print task"));**

**threadPoolExecutor.addTask(() -> System.out.println("Second print task"));**

**}**

**}**

The above program will create pool of 10 worker threads and initialize them all.

Explanation We are using two classes from standard java library in this implementation:

LinkedBlockingQueue

An optionally-bounded blocking queue based on linked nodes. This queue orders elements FIFO (first-in-first-out).

It is thread-safe in nature and acts as a temporary storage of runnable tasks that are due for execution.

Thread

All the threads get initilized and started at the creation of ThreadPoolExecutor. All threads listen on the shared workqueue for incoming tasks in never ending loop.