Practice Mock Interview →

LockSupport

Learn how to use LockSupport for building higher-level locking constructs.

*If you are interviewing, consider buying our **number#1** course for **Java Multithreading Interviews**.*

Overview

The `LockSupport` class is usually used as one of the building blocks for creating locks and synchronization classes. The class can't be instantiated and offers static methods for use. These methods of the `LockSupport` class are designed to be used as tools for creating higher-level synchronization utilities, and are not in themselves useful for most concurrency control applications. The two methods offered by this class are `park()` and `unpark()` alongwith their variations. These methods provide low-level alternatives to the deprecated methods `Thread.suspend()` and `Thread.resume()`.

`park` and `unpark` methods

The class associates a single permit with each thread that uses it. If the permit is not available the thread invoking `park()` is blocked. The blocked thread can subsequently be unblocked using the `unpark()` method by passing-in the blocked thread object as argument. If the permit is available, the thread invoking `park()` doesn't block.

The official documentation suggests the idiomatic use of the `LockSupport` class's `park()`

The official documentation suggests the idiomatic use of the `LockSupport` class's `park()` method as follows:

```
while (!canMoveForward()) {
    // ... other operations
    LockSupport.park(this);
}
```

Locking or blocking isn't used before the `while` loop or in the actions up until `park()` is invoked. Note that the `park()` method can experience spurious wake-ups and therefore, we need to check for the predicate again in a while loop.

Consider the snippet below, where the main thread unparks, i.e. makes the permit available and then parks, i.e. consumes the permit it just made available and moves forward. If the order of the two operations in the snippet is reversed, the main thread will be permanently blocked.

Java

main.java Saved

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

class Demonstration {
    public static void main( String args[] ) {

        // get a permit
        LockSupport.unpark(Thread.currentThread());
        // consume a permit
        LockSupport.park();

        System.out.println("Main thread exiting");

```

Run

Let's see a trivial example of the use of the `LockSupport` class. In the code widget below, the main thread instantiates a child thread and runs it. The child thread then parks itself by invoking the `park()` method of the `LockSupport` class. The main thread then unparks the child thread using the `unpark()` method and both threads exit.

Java

main.java Saved

```java
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.locks.LockSupport;

class Demonstration {
    public static void main( String args[] ) throws Exception {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(2);

        Thread childThread = new Thread(new Runnable() {
            @Override
            public void run() {

```

Run

FIFOLock Example

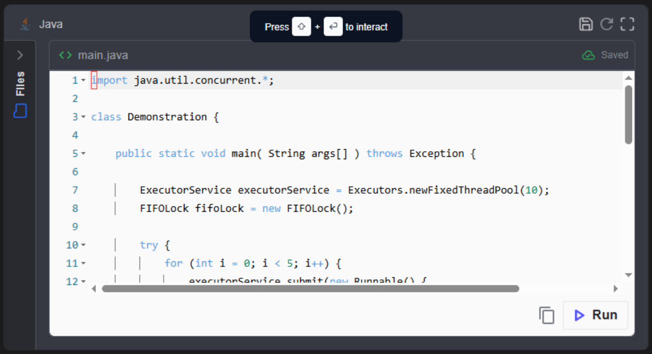
As a practical example we'll construct a FIFO lock that lets threads acquire lock in a first-in-first-out manner. Note, the lock will not be reentrant. We'll be using the `AtomicBoolean` class to maintain the status of the lock. Additionally, we'll also maintain a concurrent linked list to order the threads for lock acquisition. The `lock()` method of our `FIFOLock` class checks for two things:

1. Is the current thread requesting the lock at the head of the queue?
2. Is the lock currently unlocked?

If the above conditions satisfy then it is safe to change the status of the `FIFOLock` instance to lock and let the current thread proceed forward, otherwise the thread parks itself.

The `unlock()` method of the `FIFOLock` is relatively simple. It'll simply set the `FIFOLock` status to unlocked and then attempt to unpark the thread at the head of the queue. If there's no thread in the queue then `unpark()` has no effect.

The complete listing of the `FIFOLock`class appears below along with a test. Note that a parked thread can also be interrupted but in our implementation we haven't accounted for it to keep things simple for now.

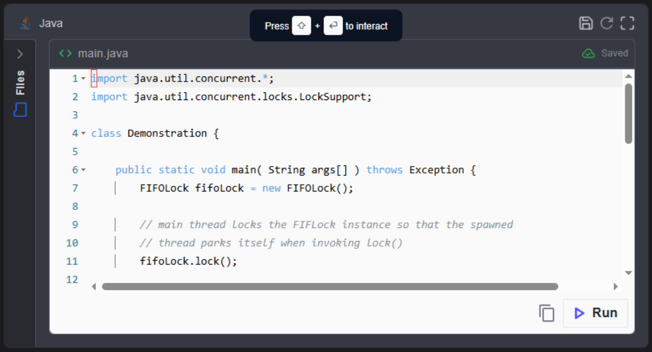


```java
import java.util.concurrent.*;

class Demonstration {

    public static void main( String args[] ) throws Exception {

        ExecutorService executorService = Executors.newFixedThreadPool(10);
        FIFOLock fifoLock = new FIFOLock();

        try {
            for (int i = 0; i < 5; i++) {
                executorService.submit(new Runnable() {
```

Blocker

The different variations of the `park()` methods also take in a parameter of type `Object` named `blocker`. The `LockSupport` class has a method `getBlocker()` that can be used to retrieve the `blocker` parameter passed-in at the time a thread invoked `park(Object blocker)`. Usually, in lock implementations the blocker parameter is the `this` object, which is the lock object that a thread is attempting to lock. This blocker object is recorded while the thread is blocked to permit monitoring and diagnostic tools to identify the reasons that threads are blocked.

In the code widget below, we reimplement our `FIFOLock` class with the `park(Object blocker)` methods. The main thread spawns a child thread that is parked, and then the main thread retrieves the blocker object associated with the child thread.



```java
import java.util.concurrent.*;
import java.util.concurrent.locks.LockSupport;

class Demonstration {

    public static void main( String args[] ) throws Exception {
        FIFOLock fifoLock = new FIFOLock();

        // main thread locks the FIFLock instance so that the spawned
        // thread parks itself when invoking lock()
        fifoLock.lock();

```

← Back lesson
Lock Interface

✓ Mark As Completed

Next →
ReentrantLock

[redacted]