educative

Thready Safety & Synchronized

This lesson explains thread-safety and the use of the synchronized keyword.

With the abstract concepts discussed, we'll now turn to the concurrency constructs offered by Java and use them in later sections to solve practical coding problems.

Thread Safe

A class and its public APIs are labelled as ***thread safe*** if multiple threads can consume the exposed APIs without causing race conditions or state corruption for the class. Note that composition of two or more thread-safe classes doesn't guarantee the resulting type to be thread-safe.

Synchronized

Java’s most fundamental construct for thread synchronization is the **`synchronized`** keyword. It can be used to restrict access to critical sections one thread at a time.

Each object in Java has an entity associated with it called the "monitor lock" or just monitor. Think of it as an exclusive lock. Once a thread gets hold of the monitor of an object, it has exclusive access to all the methods marked as synchronized. No other thread will be allowed to invoke a method on the object that is marked as synchronized and will block, till the first thread releases the monitor which is equivalent of the first thread exiting the synchronized method.

Note carefully:

1. For static methods, the monitor will be the class object, which is distinct from the monitor of each instance of the same class.
2. If an uncaught exception occurs in a synchronized method, the monitor is still released.
3. Furthermore, synchronized blocks can be re-entered.

You may think of "synchronized" as the mutex portion of a monitor.

```
class Employee {

    // shared variable
    private String name;

    // method is synchronize on 'this' object
    public synchronized void setName(String name) {
        this.name = name;
    }

    // also synchronized on the same object
    public synchronized void resetName() {

        this.name = "";
    }

    // equivalent of adding synchronized in method
    // definition
    public String getName() {
        synchronized (this) {
            return this.name;
        }
    }
}
```

As an example look at the employee class above. All the three methods are synchronized on the **"this"** object. If we created an object and three different threads attempted to execute each method of the object, only one will get access, and the other two will block. If we synchronized on a different object other than the ***this*** object, which is only possible for the `getName` method given the way we have written the code, then the 'critical sections' of the program become protected by two different locks. In that scenario, since `setName` and `resetName` would have been synchronized on the ***this*** object only one of the two methods could be executed concurrently. However `getName` would be synchronized independently of the other two methods and can be executed alongside one of them. The change would look like as follows:

```
class Employee {

    // shared variable
    private String name;
    private Object lock = new Object();

    // method is synchronize on 'this' object
    public synchronized void setName(String name) {
        this.name = name;
    }

    // also synchronized on the same object
    public synchronized void resetName() {

        this.name = "";
    }

    // equivalent of adding synchronized in method
    // definition
    public String getName() {
        // Using a different object to synchronize on
        synchronized (lock) {
            return this.name;
        }
    }
}
```

All the sections of code that you guard with synchronized blocks on the same object can have at most one thread executing inside of them at any given point in time. These sections of code may belong to different methods, classes or be spread across the code base.

Note with the use of the synchronized keyword, ***Java forces you to implicitly acquire and release the monitor-lock for the object within the same method!*** One can't explicitly acquire and release the monitor in different methods. This has an important ramification, ***the same thread will acquire and release the monitor!*** In contrast, if we used semaphore, we could acquire/release them in different methods or by different threads.

A classic newbie mistake is to synchronize on an object and then somewhere in the code reassign the object. As an example, look at the code below. We synchronize on a Boolean object in the first thread but sleep before we call `wait()` on the object. While the first thread is asleep, the second thread goes on to change the `flag`'s value. When the first thread wakes up and attempts to invoke `wait()`, it is met with a **IllegalMonitorState** exception! The object the first thread synchronized on before going to sleep has been changed, and now it is attempting to call `wait()` on an entirely different object without having synchronized on it.

```
class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        IncorrectSynchronization.runExample();
    }
}

class IncorrectSynchronization {

    Boolean flag = new Boolean(true);

    public void example() throws InterruptedException {

        Thread t1 = new Thread(new Runnable() {

            public void run() {
                synchronized (flag) {
                    try {
                        while (flag) {
                            System.out.println("First thread about to slee
                            Thread.sleep(5000);
                            System.out.println("Woke up and about to invok
                            flag.wait();
                        }
                    } catch (InterruptedException ie) {

                    }
                }
            }
```

> Marking all the methods of a class **`synchronized`** in order to make it thread-safe may reduce throughput. As a naive example, consider a class with two completely independent properties accessed by getter methods. Both the getters synchronize on the same object, and while one is being invoked, the other would be blocked because of synchronization on the same object. The solution is to lock at a finer granularity, possibly use two different locks for each property so that both can be accessed in parallel.

← Back
Moore's Law

Next →
Wait & Notify

✓ Completed

educative

Wait & Notify

wait()

> The `wait` method is exposed on each java object. Each Java object can act as a condition variable. When a thread executes the `wait` method, it releases the monitor for the object and is placed in the wait queue. ***Note that the thread must be inside a synchronized block of code that synchronizes on the same object as the one on which wait() is being called, or in other words, the thread must hold the monitor of the object on which it'll call wait.*** If not so, an illegalMonitor exception is raised!

notify()

> Like the wait method, `notify()` can only be called by the thread which owns the monitor for the object on which `notify()` is being called else an illegal monitor exception is thrown. The notify method, will awaken one of the threads in the associated wait queue, i.e., waiting on the thread's monitor.
>
> However, this thread will not be scheduled for execution immediately and will compete with other active threads that are trying to synchronize on the same object. The thread which executed notify will also need to give up the object's monitor, before any one of the competing threads can acquire the monitor and proceed forward.

notifyAll()

> This method is the same as the `notify()` one except that it wakes up all the threads that are waiting on the object's monitor.

Back
Thready Safety & Synchronized

Next
Interrupting Threads

✓ Completed

Interrupting Threads

Interrupted Exception

You'll often come across this exception being thrown from functions. When a thread wait()-s or sleep()-s then one way for it to give up waiting/sleeping is to be interrupted. If a thread is interrupted while waiting/sleeping, it'll wake up and immediately throw Interrupted exception.

The thread class exposes the `interrupt()` method which can be used to interrupt a thread that is blocked in a `sleep()` or `wait()` call. Note that invoking the interrupt method only sets a flag that is polled periodically by sleep or wait to know the current thread has been interrupted and an interrupted exception should be thrown.

Below is an example, where a thread is initially made to sleep for an hour but then interrupted by the main thread.

```
class Demonstration {

    public static void main(String args[]) throws InterruptedException {
        InterruptExample.example();
    }
}

class InterruptExample {

    static public void example() throws InterruptedException {

        final Thread sleepyThread = new Thread(new Runnable() {

            public void run() {
                try {
                    System.out.println("I am too sleepy... Let me sleep fo
17                      Thread.sleep(1000 * 60 * 60);
18                  } catch (InterruptedException ie) {
19                      System.out.println("The interrupt flag is cleard : " +
20                      Thread.currentThread().interrupt();
21                      System.out.println("Oh someone woke me up ! ");
22                      System.out.println("The interrupt flag is set now : "
23
24                  }
25              }
26          });
27
28          sleepyThread.start();
```

Take a minute to go through the output of the above program. Observe the following:

- Once the interrupted exception is thrown, the interrupt status/flag is cleared as the output of line-19 shows.
- On line-20 we again interrupt the thread and no exception is thrown. This is to emphasize that merely calling the interrupt method isn't responsible for throwing the interrupted exception. Rather the implementation should periodically check for the interrupt status and take appropriate action.
- On line 22 we print the interrupt status for the thread, which is set to true because of line 20.
- Note that there are two methods to check for the interrupt status of a thread. One is the static method `Thread.interrupted()` and the other is `Thread.currentThread().isInterrupted()`. The important difference between the two is that the static method would return the interrupt status and also clear it at the same time. On line 22 we deliberately call the object method first followed by the static method. If we reverse the ordering of the two method calls on line 22, the output for the line would be *true* and *false*, instead of *true* and *true*.

Back
Wait & Notify

Next
Volatile

✓ Completed

53% completed, meet the criteria and claim your course certificate!

Claim Certificate

Report an Issue

Ask a Question (https://discuss.educative.io/tag/interrupting-threads__multithreading-in-java__java-multithreading-for-senior-engineering-interviews)

Volatile

Volatile

The volatile concept is specific to Java. Its easier to understand volatile, if you understand the problem it solves.

If you have a variable say a counter that is being worked on by a thread, it is possible the thread keeps a copy of the counter variable in the CPU cache and manipulates it rather than writing to the main memory. The JVM will decide when to update the main memory with the value of the counter, even though other threads may read the value of the counter from the main memory and may end up reading a stale value.

If a variable is declared volatile then whenever a thread writes or reads to the volatile variable, the read and write always happen in the main memory. As a further guarantee, all the variables that are visible to the writing thread also get written-out to the main memory alongside the volatile variable. Similarly, all the variables visible to the reading thread alongside the volatile variable will have the latest values visible to the reading thread.

Volatile comes into play because of multiples levels of memory in hardware architecture required for performance enhancements. If there's a single thread that writes to the volatile variable and other threads only read the volatile variable then just using volatile is enough, however, if there's a possibility of multiple threads writing to the volatile variable then "synchronized" would be required to ensure atomic writes to the variable.

Back
Interrupting Threads

Next
Reentrant Locks & Condition Variables

✓ Completed

53% completed, meet the criteria and claim your course certificate!

Claim Certificate

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/volatile__multithreading-in-java__java-multithreading-for-senior-engineering-interviews)

educative

Reentrant Locks & Condition Variables

Reentrant Lock

Java's answer to the traditional mutex is the reentrant lock, which comes with additional bells and whistles. It is similar to the implicit monitor lock accessed when using `synchronized` methods or blocks. With the reentrant lock, you are free to lock and unlock it in different methods **but not with** different threads. If you attempt to unlock a reentrant lock object by a thread which didn't lock it initially, you'll get an **IllegalMonitorStateException**. This behavior is similar to when a thread attempts to unlock a pthread mutex.

Condition Variable

We saw how each java object exposes the three methods, `wait()`,`notify()` and `notifyAll()` which can be used to suspend threads till some condition becomes true. You can think of Condition as factoring out these three methods of the object monitor into separate objects so that there can be multiple wait-sets per object. As a reentrant lock replaces `synchronized` blocks or methods, a condition replaces the object monitor methods. In the same vein, one can't invoke the condition variable's methods without acquiring the associated lock, just like one can't wait on an object's monitor without synchronizing on the object first. In fact, a reentrant lock exposes an API to create new condition variables, like so:

```
Lock lock = new ReentrantLock();
Condition myCondition  = lock.newCondition();
```

Notice, how can we now have multiple condition variables associated with the same lock. In the **`synchronized`** paradigm, we could only have one wait-set associated with each object.

java.util.concurrent

Java's util.concurrent package provides several classes that can be used for solving everyday concurrency problems and should always be preferred than reinventing the wheel. Its offerings include thread-safe data structures such as **`ConcurrentHashMap`**.

← Back

Volatile

Next →

Missing Signals

✓ Completed

53% completed, meet the criteria and claim your course certificate!

Claim Certificate

Report an Issue

Ask a Question (https://discuss.educative.io/tag/reentrant-locks--condition-variables__multithreading-in-java__java-multithreading-for-senior-engineering-interviews)

educative

Missed Signals

Missed Signals

A missed signal happens when a signal is sent by a thread before the other thread starts waiting on a condition. This is exemplified by the following code snippet. Missed signals are caused by using the wrong concurrency constructs. In the example below, a condition variable is used to coordinate between the **signaller** and the **waiter** thread. The condition is signaled at a time when no thread is waiting on it causing a missed signal.

In later sections, you'll learn that the way we are using the condition variable's `await` method is incorrect. The idiomatic way of using `await` is in a while loop with an associated boolean condition. For now, observe the possibility of losing signals between threads.

```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

class Demonstration {

    public static void main(String args[]) throws InterruptedException {
        MissedSignalExample.example();
    }
}

class MissedSignalExample {

    public static void example() throws InterruptedException {

        final ReentrantLock lock = new ReentrantLock();
        final Condition condition = lock.newCondition();

        Thread signaller = new Thread(new Runnable() {

            public void run() {
                lock.lock();
                condition.signal();
                System.out.println("Sent signal");
                lock.unlock();
            }
        });

        Thread waiter = new Thread(new Runnable() {
```

Missed Signal Example

The above code when ran, will never print the statement **`Program Exiting`** and execution would time out. Apart from refactoring the code to match the idiomatic usage of condition variables in a while loop, the other possible fix is to use a **semaphore** for signalling between the two threads as shown below

```
import java.util.concurrent.Semaphore;

class Demonstration {

    public static void main(String args[]) throws InterruptedException {
        FixedMissedSignalExample.example();
    }
}

class FixedMissedSignalExample {

    public static void example() throws InterruptedException {

        final Semaphore semaphore = new Semaphore(1);

        Thread signaller = new Thread(new Runnable() {

            public void run() {
                semaphore.release();
                System.out.println("Sent signal");
            }
        });

        Thread waiter = new Thread(new Runnable() {
```

```
        Thread waiter = new Thread(new Runnable() {

            public void run() {
                try {
                    semaphore.acquire();
```

Fixed Missed Signal

Back

Reentrant Locks & Condition Variables

Next

Semaphore in Java

✓ Completed

53% completed, meet the criteria and claim your course certificate!

Claim Certificate

Report an Issue

Ask a Question (https://discuss.educative.io/tag/missed-signals__multithreading-in-java__java-multithreading-for-senior-engineering-interviews)

educative

Semaphore in Java

Semaphore

Java's semaphore can be **`releas()-ed`** or **`acquire()-d`** for signalling amongst threads. However the important call out when using semaphores is to make sure that the permits acquired should equal permits returned. Take a look at the following example, where a runtime exception causes a deadlock.

```
import java.util.concurrent.Semaphore;

class Demonstration {

    public static void main(String args[]) throws InterruptedException {
        IncorrectSemaphoreExample.example();
    }
}

class IncorrectSemaphoreExample {

    public static void example() throws InterruptedException {

        final Semaphore semaphore = new Semaphore(1);

        Thread badThread = new Thread(new Runnable() {

            public void run() {

                while (true) {

                    try {
                        semaphore.acquire();
                    } catch (InterruptedException ie) {
                        // handle thread interruption
                    }

                    // Thread was meant to run forever but runs into an
```

Incorrect Use of Semaphore

The above code when run would time out and show that one of the threads threw an exception. The code is never able to release the semaphore causing the other thread to block forever. Whenever using locks or semaphores, remember to unlock or release the semaphore in a **finally** block. The corrected version appears below.

```java
import java.util.concurrent.Semaphore;

class Demonstration {

    public static void main(String args[]) throws InterruptedException {
        CorrectSemaphoreExample.example();
    }
}

class CorrectSemaphoreExample {

    public static void example() throws InterruptedException {

        final Semaphore semaphore = new Semaphore(1);

        Thread badThread = new Thread(new Runnable() {

            public void run() {

                while (true) {

                    try {
                        semaphore.acquire();
                        try {
                            throw new RuntimeException("");
                        } catch (Exception e) {
                            // handle any program logic exception and exit
                            return;
```

Running the above code will print the `Exiting Program` statement.

← Back

Missed Signals

Next →

Spurious Wakeups

✓ Completed

53% completed, meet the criteria and claim your course certificate!

Claim Certificate

Report an Issue

Ask a Question (https://discuss.educative.io/tag/semaphore-in-java__multithreading-in-java__java-multithreading-for-senior-engineering-interviews)

Spurious Wakeups

Spurious Wakeups

Spurious mean **fake** or **false**. A spurious wakeup means a thread is woken up even though no signal has been received. Spurious wakeups are a reality and are one of the reasons why the pattern for waiting on a condition variable happens in a while loop as discussed in earlier chapters. There are technical reasons beyond our current scope as to why spurious wakeups happen, but for the curious on POSIX based operating systems when a process is signaled, all its waiting threads are woken up. Below comment is a directly lifted from Java's documentation for the `wait(long timeout)` method.

```
* A thread can also wake up without being notified, interrup
ted, or
* timing out, a so-called <i>spurious wakeup</i>.  While thi
s will rarely
* occur in practice, applications must guard against it by t
esting for
* the condition that should have caused the thread to be awa
kened and
* continuing to wait if the condition is not satisfied.  I
n other words,
* waits should always occur in loops, like this one:
*
*     synchronized (obj) {
*         while (condition does not hold)
*             obj.wait(timeout);
*         ... // Perform action appropriate to condition
*     }
*
```

Back

Semaphore in Java

Next

Miscellaneous Topics

✓ Completed

53% completed, meet the criteria and claim your course certificate!

Claim Certificate

Report an Issue

Ask a Question (https://discuss.educative.io/tag/spurious-wakeups__multithreading-in-java__java-multithreading-for-senior-engineering-interviews)

Miscellaneous Topics

Lock Fairness

We'll briefly touch on the topic of fairness in locks since its out of scope for this course. When locks get acquired by threads, there's no guarantee of the order in which threads are granted access to a lock. A thread requesting lock access more frequently may be able to acquire the lock unfairly greater number of times than other locks. Java locks can be turned into fair locks by passing in the fair constructor parameter. However, fair locks exhibit lower throughput and are slower compared to their unfair counterparts.

Thread Pools

Imagine an application that creates threads to undertake short-lived tasks. The application would incur a performance penalty for first creating hundreds of threads and then tearing down the allocated resources for each thread at the ends of its life. The general way programming frameworks solve this problem is by creating a pool of threads, which are handed out to execute each concurrent task and once completed, the thread is returned to the pool

Java offers thread pools via its **Executor Framework**. The framework includes classes such as the `ThreadPoolExecutor` for creating thread pools.

← Back
Spurious Wakeups

Next →
Java Memory Model

✓ Completed

53% completed, meet the criteria and claim your course certificate!

Claim Certificate

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/miscellaneous-topics__multithreading-in-java__java-multithreading-for-senior-engineering-interviews)