

Introduction

This lesson details the reasons why threads exist and what benefit do they provide. We also discuss the problems that come with threads.

We'll cover the following

- [What good is concurrency?](#)
- [Benefits of Threads](#)
- [Performance Gains via Multi-Threading](#)
- [Problems with Threads](#)

What good is concurrency?#

Understanding of how threading works and knowledge of concurrent programming principles exhibit maturity and technical depth of a candidate and can be an important differentiator in landing a higher leveling offer at a company. First, we have to understand why threading models exist and what good do they provide?

Threads like most computer science concepts aren't physical objects. The closest tangible manifestation of threads can be seen in a debugger. The screen-shot below, shows the threads of our program suspended in the debugger.

Suspended threads in a debugger

The simplest example to think of a concurrent system is a single-processor machine running your favorite IDE. Say you edit one of your code files and click save, that clicking of the button will initiate a workflow which will cause bytes to be written out to the underlying physical disk. However, IO is an expensive operation, and the CPU will be idle while bytes are being written out to the disk.

Whilst IO takes place, the idle CPU could work on something useful and here is where threads come in - the IO thread is **switched out** and the UI thread gets scheduled on the CPU so that if you click elsewhere on the screen, your IDE is still responsive and does not appear hung or frozen.

Threads can give the illusion of multitasking even though at any given point in time the CPU is executing only one thread. Each thread gets a slice of time on the CPU and then gets switched out either because it initiates a task which requires waiting and not utilizing the CPU or it completes its time slot on the CPU. There are much more

nuances and intricacies on how thread scheduling works but what we just described, forms the basis of it.

With advances in hardware technology, it is now common to have multi-core machines. Applications can take advantage of these architectures and have a dedicated CPU run each thread.

Benefits of Threads#

1. Higher throughput, though in some pathetic scenarios it is possible to have the overhead of context switching among threads steal away any throughput gains and result in worse performance than a single-threaded scenario. However such cases are unlikely and an exception, rather than the norm.
2. Responsive applications that give the illusion of multi-tasking.
3. Efficient utilization of resources. Note that thread creation is light-weight in comparison to spawning a brand new process. Web servers that use threads instead of creating new processes when fielding web requests, consume far fewer resources.

All other benefits of multi-threading are extensions of or indirect benefits of the above.

Performance Gains via Multi-Threading#

As a concrete example, consider the example code below. The task is to **compute the sum of all the integers from o to Integer.MAX_VALUE**. In the first scenario, we have a single thread doing the summation while in the second scenario we split the range into two parts and have one thread sum for each range. In the end, we add the two half sums to get the combined sum. We measure the time taken for each scenario and print it.

```
12  
13  
14  
15  
16  
  
class Demonstration {  
    public static void main( String args[] ) throws InterruptedException  
    {  
        SumUpExample.runTest();  
    }  
}
```

```

class SumUpExample {
    long startRange;
    long endRange;
    long counter = 0;
    static long MAX_NUM = Integer.MAX_VALUE;
    public SumUpExample(long startRange, long endRange) {
        this.startRange = startRange;
        this.endRange = endRange;
    }
    public void add() {
        for (long i = startRange; i <= endRange; i++) {
            counter += i;
        }
    }
    static public void twoThreads() throws InterruptedException {
        long start = System.currentTimeMillis();
        SumUpExample s1 = new SumUpExample(1, MAX_NUM / 2);
        SumUpExample s2 = new SumUpExample(1 + (MAX_NUM / 2),
        MAX_NUM);

        Thread t1 = new Thread(() -> {
            s1.add();
        });

        Thread t2 = new Thread(() -> {
            s2.add();
        });

        t1.start();
        t2.start();
    }
}

```

In my run, I see the two threads scenario run within **652 milliseconds** whereas the single thread scenario runs in **886 milliseconds**. You may observe different numbers but the time taken by two threads would always be less than the time taken by a single thread. The performance gains can be many folds depending on the availability of multiple CPUs and the nature of the problem being solved. However, there will always be problems that don't yield

well to a multi-threaded approach and may very well be solved efficiently using a single thread.

Problems with Threads#

However, as it is said, there's no free lunch in life. The premium for using threads manifests in the following forms:

1. **Usually very hard to find bugs**, some that may only rear head in production environments
2. **Higher cost of code maintenance** since the code inherently becomes harder to reason about
3. **Increased utilization of system resources**. Creation of each thread consumes additional memory, CPU cycles for book-keeping and waste of time in context switches.
4. **Programs may experience slowdown** as coordination amongst threads comes at a price. Acquiring and releasing locks adds to program execution time. Threads fighting over acquiring locks cause lock contention.

With this backdrop lets delve into more details of concurrent programming about which you are likely to be quizzed in an interview.

Program vs Process vs Thread

This lesson discusses the differences between a program, process and a thread. Also included is an example of a thread-unsafe program.

We'll cover the following

- [Program](#)
- [Process](#)
- [Thread](#)
- [Caveats](#)
- [Counter Program](#)
- [Thread unsafe class](#)

Program#

A program is a set of instructions and associated data that resides on the disk and is loaded by the operating system to perform some task. An executable file or a python script file are examples of programs. In order to run a program, the operating system's kernel is

first asked to create a new process, which is an environment in which a program executes.

Process

A process is a program in execution. A process is an execution environment that consists of instructions, user-data, and system-data segments, as well as lots of other resources such as CPU, memory, address-space, disk and network I/O acquired at runtime. A program can have several copies of it running at the same time but a process necessarily belongs to only one program.

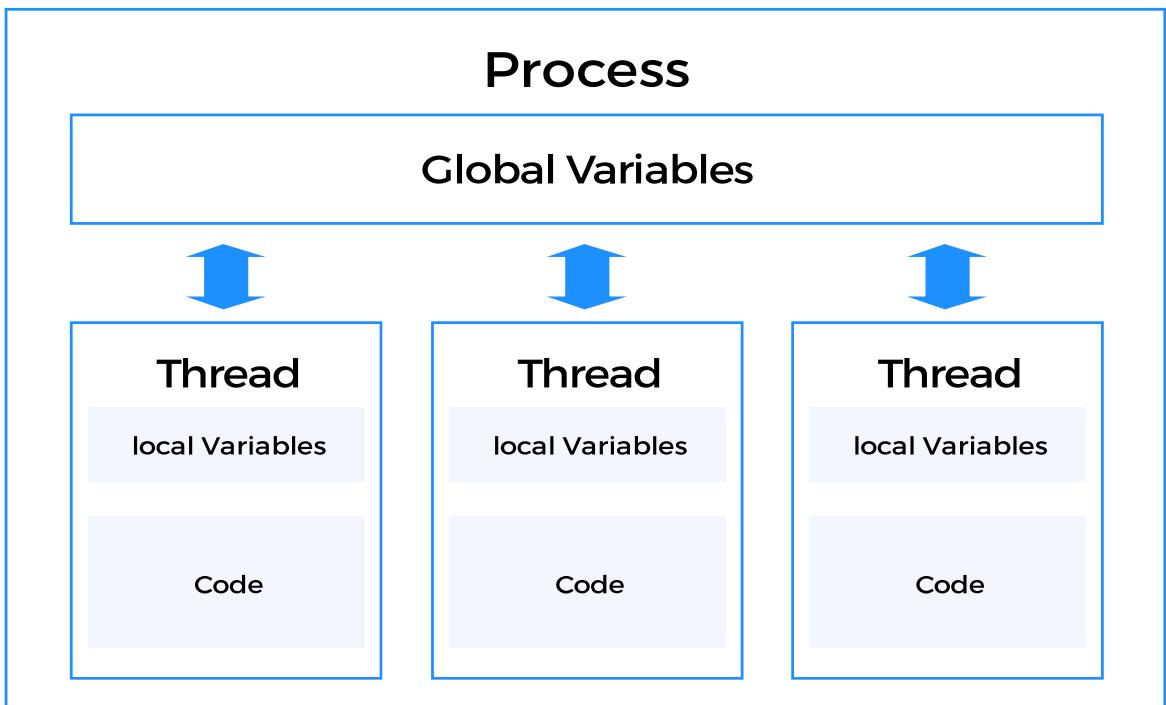
Thread

Thread is the smallest unit of execution in a process. A thread simply executes instructions serially. A process can have multiple threads running as part of it. Usually, there would be some state associated with the process that is shared among all the threads and in turn each thread would have some state private to itself. The globally shared state amongst the threads of a process is visible and accessible to all the threads, and special attention needs to be paid when any thread tries to read or write to this global shared state. There are several constructs offered by various programming languages to guard and discipline the access to this global state, which we will go into further detail in upcoming lessons.

Caveats

Note a program or a process are often used interchangeably but most of the times the intent is to refer to a process.

There's also the concept of "multiprocessing" systems, where multiple processes get scheduled on more than one CPU. Usually, this requires hardware support where a single system comes with multiple cores or the execution takes place in a cluster of machines. Processes don't share any resources amongst themselves whereas threads of a process can share the resources allocated to that particular process, including memory address space. However, languages do provide facilities to enable inter-process communication.



Counter Program#

Below is an example highlighting how multi-threading necessitates caution when accessing shared data amongst threads. Incorrect synchronization between threads can lead to wildly varying program outputs depending on in which order threads get executed.

Consider the below snippet of code

```

1. int counter = 0;
2.
3. void incrementCounter() {
4.   counter++;
5. }
```

The increment on **line 4** is likely to be decompiled into the following steps on a computer:

- Read the value of the variable counter from the register where it is stored
- Add one to the value just read
- Store the newly computed value back to the register

The innocuous looking statement on **line 4** is really a three step process!

Now imagine if we have two threads trying to execute the same function **incrementCounter** then one of the ways the execution of the two threads can take place is as follows:

Lets call one thread as **T1** and the other as **T2**. Say the counter value is equal to 7.

1. **T1** is currently scheduled on the CPU and enters the function. It performs step A i.e. reads the value of the variable from the register, which is 7.
2. The operating system decides to context switch **T1** and bring in **T2**.
3. **T2** gets scheduled and luckily gets to complete all the three steps **A**, **B** and **C** before getting switched out for **T1**. It reads the value 7, adds one to it and stores 8 back.
4. **T1** comes back and since its state was saved by the operating system, it still has the stale value of 7 that it read before being context switched. It doesn't know that behind its back the value of the variable has been updated. It unfortunately thinks the value is still 7, adds one to it and overwrites the existing 8 with its own computed 8. If the threads executed serially the final value would have been 9.

The problems should be apparent to the astute reader. Without properly guarding access to mutable variables or data-structures, threads can cause hard to find bugs.

Since the execution of the threads can't be predicted and is entirely up to the operating system, we can't make any assumptions about the order in which threads get scheduled and executed.

Thread unsafe class#

Take a minute to go through the following program. It increments a counter and decrements it an equal number of times. The final value of the counter should be zero, however, if you run the program enough times, you'll sometimes get the correct zero value, and at others, you'll get a non-zero value. We sleep the threads to give them a chance to run in a non-deterministic order.

```
import java.util.Random;

class DemoThreadUnsafe {

    // We'll use this to randomly sleep our threads
    static Random random = new
Random(System.currentTimeMillis());

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

        // create object of unsafe counter
        ThreadUnsafeCounter badCounter = new
ThreadUnsafeCounter();

        // setup thread1 to increment the badCounter 200 times
        Thread thread1 = new Thread(new Runnable() {

            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    badCounter.increment();
                    DemoThreadUnsafe.sleepRandomlyForLessThan10Secs();
                }
            }
        });

        // setup thread2 to decrement the badCounter 200 times
        Thread thread2 = new Thread(new Runnable() {

            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    badCounter.decrement();
                    DemoThreadUnsafe.sleepRandomlyForLessThan10Secs();
                }
            }
        });

        // run both threads
        thread1.start();
        thread2.start();

        // wait for t1 and t2 to complete.
        thread1.join();
    }
}
```

```

        thread2.join();

        // print final value of counter
        badCounter.printFinalCounterValue();
    }

    public static void sleepRandomlyForLessThan10Secs() {
        try {
            Thread.sleep(random.nextInt(10));
        } catch (InterruptedException ie) {
        }
    }
}

class ThreadUnsafeCounter {

    int count = 0;

    public void increment() {
        count++;
    }

    public void decrement() {
        count--;
    }

    void printFinalCounterValue() {
        System.out.println("counter is: " + count);
    }
}

```

Concurrency vs Parallelism

This lesson clarifies the common misunderstandings and confusions around concurrency and parallelism.

We'll cover the following

- [Introduction](#)
- [Serial Execution](#)
- [Concurrency](#)
- [Parallelism](#)
- [Concurrency vs Parallelism](#)

[Introduction #](#)

Concurrency and *Parallelism* are often confused to refer to the ability of a system to run multiple distinct programs at the same time. Though the two terms are somewhat related yet they mean very different things. To clarify the concept, we'll borrow a juggler from a circus to use as an analogy. Consider the juggler to be a machine and the balls he juggles as processes.

Serial Execution#

When programs are serially executed, they are scheduled one at a time on the CPU. Once a task gets completed, the next one gets a chance to run. Each task is run from the beginning to the end without interruption. The analogy for serial execution is a circus juggler who can only juggle one ball at a time. Definitely not very entertaining!

Concurrency#

A concurrent program is one that can be decomposed into constituent parts and each part can be executed out of order or in partial order without affecting the final outcome. A system capable of running several distinct programs or more than one independent unit of the same program in overlapping time intervals is called a concurrent system. The execution of two programs or units of the same program may not happen simultaneously.

A concurrent system can have two programs *in progress* at the same time where *progress* doesn't imply execution. One program can be suspended while the other executes. Both programs are able to make progress as their execution is interleaved. In concurrent systems, the goal is to maximize throughput and minimize latency. For example, a browser running on a single core machine has to be responsive to user clicks but also be able to render HTML on screen as quickly as possible. Concurrent systems achieve lower latency and higher throughput when programs running on the system require frequent network or disk I/O.

The classic example of a concurrent system is that of an operating system running on a single core machine. Such an operating system is concurrent but not parallel. It can only process one task at any given point in time but all the tasks being managed by the operating system appear to make progress because the operating system is designed for concurrency. Each task gets a slice of the CPU time to execute and move forward.

Going back to our circus analogy, a concurrent juggler is one who can juggle several balls at the same time. However, at any one point in

time, he can only have a single ball in his hand while the rest are in flight. Each ball gets a time slice during which it lands in the juggler's hand and then is thrown back up. A concurrent system is in a similar sense *juggling* several processes at the same time.

Parallelism

A parallel system is one which necessarily has the ability to execute multiple programs **at the same time**. Usually, this capability is aided by hardware in the form of multicore processors on individual machines or as computing clusters where several machines are hooked up to solve independent pieces of a problem simultaneously. Remember an individual problem has to be concurrent in nature, that is portions of it can be worked on independently without affecting the final outcome before it can be executed in parallel.

In parallel systems the emphasis is on increasing throughput and optimizing usage of hardware resources. The goal is to extract out as much computation speedup as possible. Example problems include matrix multiplication, 3D rendering, data analysis, and particle simulation.

Revisiting our juggler analogy, a parallel system would map to at least two or more jugglers juggling one or more balls. In the case of an operating system, if it runs on a machine with say four CPUs then the operating system can execute four tasks at the same time, making execution parallel. Either a single (large) problem can be executed in parallel or distinct programs can be executed in parallel on a system supporting parallel execution.

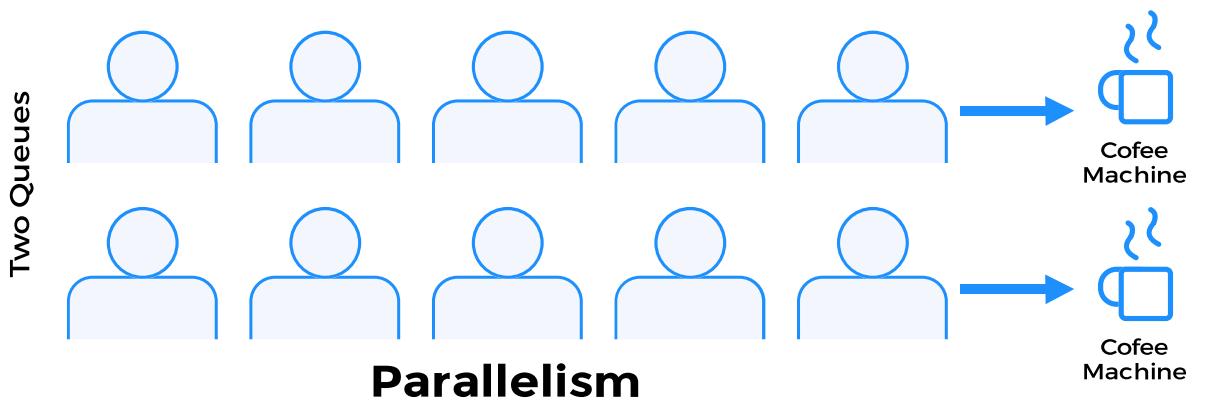
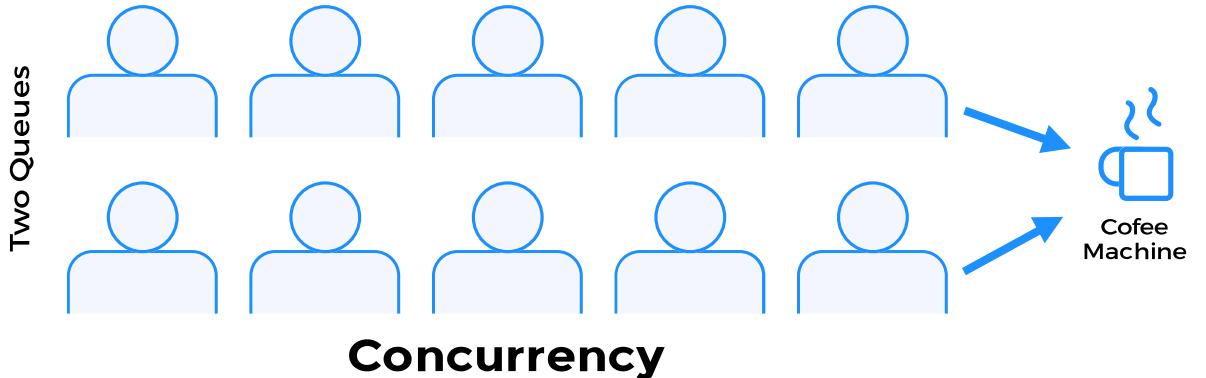
Concurrency vs Parallelism

From the above discussion it should be apparent that a concurrent system need not be parallel, whereas a parallel system is indeed concurrent. Additionally, a system can be both concurrent and parallel e.g. a multitasking operating system running on a multicore machine.

Concurrency is about *dealing with lots of things at once*. Parallelism is about *doing lots of things at once*. Last but not the least, you'll find literature describing concurrency as a property of a program or a system whereas parallelism as a runtime behaviour of executing multiple tasks.

We end the lesson with an analogy, frequently quoted in online literature, of customers waiting in two queues to buy coffee. Single-

processor concurrency is akin to alternatively serving customers from the two queues but with a single coffee machine, while parallelism is similar to serving each customer queue with a dedicated coffee machine.



Cooperative Multitasking vs Preemptive Multitasking

This lesson details the differences between the two common models of multitasking.

We'll cover the following

- [Introduction](#)
- [Preemptive Multitasking](#)
- [Cooperative Multitasking](#)
- [Cooperative vs Preemptive](#)

[Introduction #](#)

A system can achieve concurrency by employing one of the following multitasking models:

- Preemptive Multitasking
- Cooperative Multitasking

Preemptive Multitasking #

In preemptive multitasking, the operating system preempts a program to allow another waiting task to run on the CPU. Programs or threads can't decide how long for or when they can use the CPU. The operating system's scheduler decides which thread or program gets to use the CPU next and for how much time. Furthermore, scheduling of programs or threads on the CPU isn't predictable. A thread or program once taken off of the CPU by the scheduler can't determine when it will get on the CPU next. As a consequence, if a malicious program initiates an infinite loop, it only hurts itself without affecting other programs or threads. Lastly, the programmer isn't burdened to decide when to give up control back to the CPU in code.

Cooperative Multitasking #

Cooperative Multitasking involves well-behaved programs to voluntarily give up control back to the scheduler so that another program can run. A program or thread may give up control after a period of time has expired or if it becomes idle or logically blocked. The operating system's scheduler has no say in how long a program or thread runs for. A malicious program can bring the entire system to a halt by busy waiting or running an infinite loop and not giving up control. The process scheduler for an operating system implementing cooperative multitasking is called a cooperative scheduler. As the name implies, the participating programs or threads are required to cooperate to make the scheduling scheme work.

Cooperative vs Preemptive #

Early versions of both Windows and Mac OS used cooperative multitasking. Later on preemptive multitasking was introduced in Windows NT 3.1 and in Mac OS X. However, preemptive multitasking has always been a core feature of Unix based systems.

Synchronous vs Asynchronous

This lesson discusses the differences between asynchronous and synchronous programming which are often talked about in the context of concurrency.

We'll cover the following

- [Synchronous](#)
- [Asynchronous](#)

Synchronous#

Synchronous execution refers to line-by-line execution of code. If a function is invoked, the program execution waits until the function call is completed. Synchronous execution blocks at each method call before proceeding to the next line of code. A program executes in the same sequence as the code in the source code file. Synchronous execution is synonymous to serial execution.

Asynchronous#

Asynchronous (or `async`) execution refers to execution that doesn't block when invoking subroutines. Or if you prefer the more fancy Wikipedia definition: *Asynchronous programming is a means of parallel programming in which a unit of work runs separately from the main application thread and notifies the calling thread of its completion, failure or progress.* An asynchronous program doesn't wait for a task to complete and can move on to the next task.

In contrast to synchronous execution, asynchronous execution doesn't necessarily execute code line by line, that is instructions may not run in the sequence they appear in the code. Async execution can invoke a method and move onto the next line of code without waiting for the invoked function to complete or receive its result. Usually, such methods return an entity sometimes called a **future** or **promise** that is a representation of an in-progress computation. The program can query for the status of the computation via the returned future or promise and retrieve the result once completed. Another pattern is to pass a callback function to the asynchronous function call which is invoked with the results when the asynchronous function is done processing. Asynchronous programming is an excellent choice for applications that do extensive network or disk I/O and spend most of their time waiting. As an example, Javascript enables concurrency using AJAX library's asynchronous method calls. In non-threaded environments, asynchronous programming provides an alternative to threads in

order to achieve concurrency and fall under the cooperative multitasking model.

Asynchronous programming support in Java has become a lot more robust starting with Java 8, however, the topic is out of scope for this course so we only mention it in passing.

I/O Bound vs CPU Bound

We delve into the characteristics of programs with different resource-use profiles and how that can affect program design choices.

We'll cover the following

- [I/O Bound vs CPU Bound](#)
- [CPU Bound](#)
- [I/O Bound](#)
- [Caveats](#) 

I/O Bound vs CPU Bound

We write programs to solve problems. Programs utilize various resources of the computer systems on which they run. For instance a program running on your machine will broadly require:

- CPU Time
- Memory
- Networking Resources
- Disk Storage

Depending on what a program does, it can require heavier use of one or more resources. For instance, a program that loads gigabytes of data from storage into main memory would **hog** the main memory of the machine it runs on. Another can be writing several gigabytes to permanent storage, requiring abnormally high disk i/o.

CPU Bound

Programs which are **compute-intensive** i.e. program execution requires very high utilization of the CPU (close to 100%) are called CPU bound programs. Such programs primarily depend on improving CPU speed to decrease program completion time. This could include programs such as data crunching, image processing, matrix multiplication etc.

If a CPU bound program is provided a more powerful CPU it can potentially complete faster. Eventually, there is a limit on how powerful a single CPU can be. At this point, the recourse is to harness the computing power of multiple CPUs and structure your program code in a way that can take advantage of the multiple CPU units available. Say we are trying to sum up the first 1 million natural numbers. A single-threaded program would sum in a single loop from 1 to 1000000. To cut down on execution time, we can create two threads and divide the range into two halves. The first thread sums the numbers from 1 to 500000 and the second sums the numbers from 500001 to 1000000. **If there are two processors available on the machine, then each thread can independently run on a single CPU in parallel.** In the end, we sum the results from the two threads to get the final result. Theoretically, the multithreaded program should finish in half the time that it takes for the single-threaded program. However, there will be a slight overhead of creating the two threads and merging the results from the two threads.

Multithreaded programs can improve performance in cases where the problem lends itself to being divided into smaller pieces that different threads can work on independently. This may not always be true though.

I/O Bound[#]

I/O bound programs are the opposite of CPU bound programs. Such programs spend most of their time waiting for input or output operations to complete while the CPU sits idle. I/O operations can consist of operations that write or read from main memory or network interfaces. Because the CPU and main memory are physically separate a data bus exists between the two to transfer bits to and fro. Similarly, data needs to be moved between network interfaces and CPU/memory. Even though the physical distances are tiny, the time taken to move the data across is big enough for several thousand CPU cycles to go waste. This is why I/O bound programs would show relatively lower CPU utilization than CPU bound programs.

Caveats[#]

Both types of programs can benefit from concurrent architectures. If a program is CPU bound we can increase the number of processors and structure our program to spawn multiple threads that individually run on a dedicated or shared CPU. For I/O bound programs, it makes sense to have a thread give up CPU control if it is waiting for an I/O operation to complete so that another thread can

get scheduled on the CPU and utilize CPU cycles. Different programming languages come with varying support for multithreading. For instance, Javascript is single-threaded, Java provides full-blown multithreading and Python is sort of multithreaded as it can only have a single thread in running state because of its global interpreter lock (GIL) limitation. However, all three languages support asynchronous programming models which is another way for programs to be concurrent (but not parallel).

For completeness we should mention that there are also memory-bound programs that depend on the amount of memory available to speed up execution.

Throughput vs Latency

This lesson discusses throughput and latency in the context of concurrent systems.

We'll cover the following

- [Throughput](#)
- [Latency](#)
- [Throughput vs Latency](#)

Throughput#

Throughput is defined as the *rate of doing work* or how much work gets done per unit of time. If you are an Instagram user, you could define throughput as the number of images your phone or browser downloads per unit of time.

Latency#

Latency is defined as the *time required to complete a task or produce a result*. Latency is also referred to as *response time*. The time it takes for a web browser to download Instagram images from the internet is the latency for downloading the images.

Throughput vs Latency#

The two terms are more frequently used when describing networking links and have more precise meanings in that domain. In the context of concurrency, throughput can be thought of as time taken to execute a program or computation. For instance, imagine a program that is given

hundreds of files containing integers and asked to sum up all the numbers. Since addition is commutative each file can be worked on in parallel. In a single-threaded environment, each file will be sequentially processed but in a concurrent system, several threads can work in parallel on distinct files. Of course, there will be some overhead to manage the state including already processed files. However, such a program will complete the task much faster than a single thread. The performance difference will become more and more apparent as the number of input files increases. The throughput in this example can be defined as the number of files processed by the program in a minute. And latency can be defined as the total time taken to completely process all the files. As you observe in a multithreaded implementation throughput will go up and latency will go down. More work gets done in less amount of time. In general, the two have an inverse relationship.

Critical Sections & Race Conditions

This section exhibits how incorrect synchronization in a critical section can lead to race conditions and buggy code. The concepts of critical section and race condition are explained in depth. Also included is an executable example of a race condition.

We'll cover the following

- o [Critical Section](#)
- o [Race Condition](#)
- o [Example Thread Race](#)

A program is a set of instructions being executed, and multiple threads of a program can be executing different sections of the program code. However, caution should be exercised when threads of the same program attempt to execute the same portion of code as explained in the following paragraphs.

Critical Section#

Critical section is any piece of code that has the possibility of being executed concurrently by more than one thread of the application and exposes any shared data or resources used by the application for access.

Race Condition#

Race conditions happen when threads run through critical sections without thread synchronization. The threads "race" through the critical section to write or read shared resources and depending on the order in which threads finish the "race", the program output changes. In a race condition,

threads access shared resources or program variables that might be worked on by other threads at the same time causing the application data to be inconsistent.

As an example consider a thread that tests for a state/condition, called a predicate, and then based on the condition takes subsequent action. This sequence is called **test-then-act**. The pitfall here is that the state can be mutated by the second thread just after the test by the first thread and before the first thread takes action based on the test. A different thread changes the predicate in between the **test and act**. In this case, action by the first thread is not justified since the predicate doesn't hold when the action is executed.

Consider the snippet below. We have two threads working on the same variable `randInt`. The modifier thread perpetually updates the value of `randInt` in a loop while the printer thread prints the value of `randInt` only if `randInt` is divisible by 5. If you let this program run, you'll notice some values get printed even though they aren't divisible by 5 demonstrating a thread unsafe version of **test-then-act**.

Example Thread Race#

The below program spawns two threads. One thread prints the value of a shared variable whenever the shared variable is divisible by 5. A race condition happens when the printer thread executes a *test-then-act* if clause, which checks if the shared variable is divisible by 5 but before the thread can print the variable out, its value is changed by the modifier thread. Some of the printed values aren't divisible by 5 which verifies the existence of a race condition in the code.

```
import java.util.*;

class Demonstration {

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

class RaceCondition {

    int randInt;
    Random random = new Random(System.currentTimeMillis());

    void printer() {
```

```
int i = 1000000;
while (i != 0) {
    if (randInt % 5 == 0) {
        if (randInt % 5 != 0)
            System.out.println(randInt);
    }
    i--;
}
}

void modifier() {

    int i = 1000000;
    while (i != 0) {
        randInt = random.nextInt(1000);
        i--;
    }
}

public static void runTest() throws InterruptedException {

    final RaceCondition rc = new RaceCondition();
    Thread thread1 = new Thread(new Runnable() {

        @Override
        public void run() {
            rc.printer();
        }
    });
    Thread thread2 = new Thread(new Runnable() {

        @Override
        public void run() {
            rc.modifier();
        }
    });

    thread1.start();
    thread2.start();

    thread1.join();
    thread2.join();
}
```

Even though the if condition on **line 19** makes a check for a value which is divisible by 5 and only then prints `randInt`. It is just after the if check and before the print statement i.e. in-between **lines 19** and **21**, that the value of `randInt` is modified by the modifier thread! This is what constitutes a race condition.

For the impatient, the fix is presented below where we guard the read and write of the `randInt` variable using the `RaceCondition` object as the monitor. Don't fret if the solution doesn't make sense for now, it would, once we cover various topics in the lessons ahead.

```
import java.util.*;  
  
class Demonstration {  
  
    public static void main(String args[]) throws InterruptedException {  
        RaceCondition.runTest();  
    }  
}  
  
class RaceCondition {  
  
    int randInt;  
    Random random = new Random(System.currentTimeMillis());  
  
    void printer() {  
  
        int i = 1000000;  
        while (i != 0) {  
            synchronized(this) {  
                if (randInt % 5 == 0) {  
                    if (randInt % 5 != 0)  
                        System.out.println(randInt);  
                }  
            }  
            i--;  
        }  
    }  
  
    void modifier() {  
  
        int i = 1000000;  
        while (i != 0) {  
            synchronized(this) {  
                randInt = random.nextInt(1000);  
                i--;  
            }  
        }  
    }  
}
```

```
}
```

```
public static void runTest() throws InterruptedException {
```

```
    final RaceCondition rc = new RaceCondition();  
    Thread thread1 = new Thread(new Runnable() {
```

```
        @Override  
        public void run() {  
            rc.printer();  
        }  
    });
```

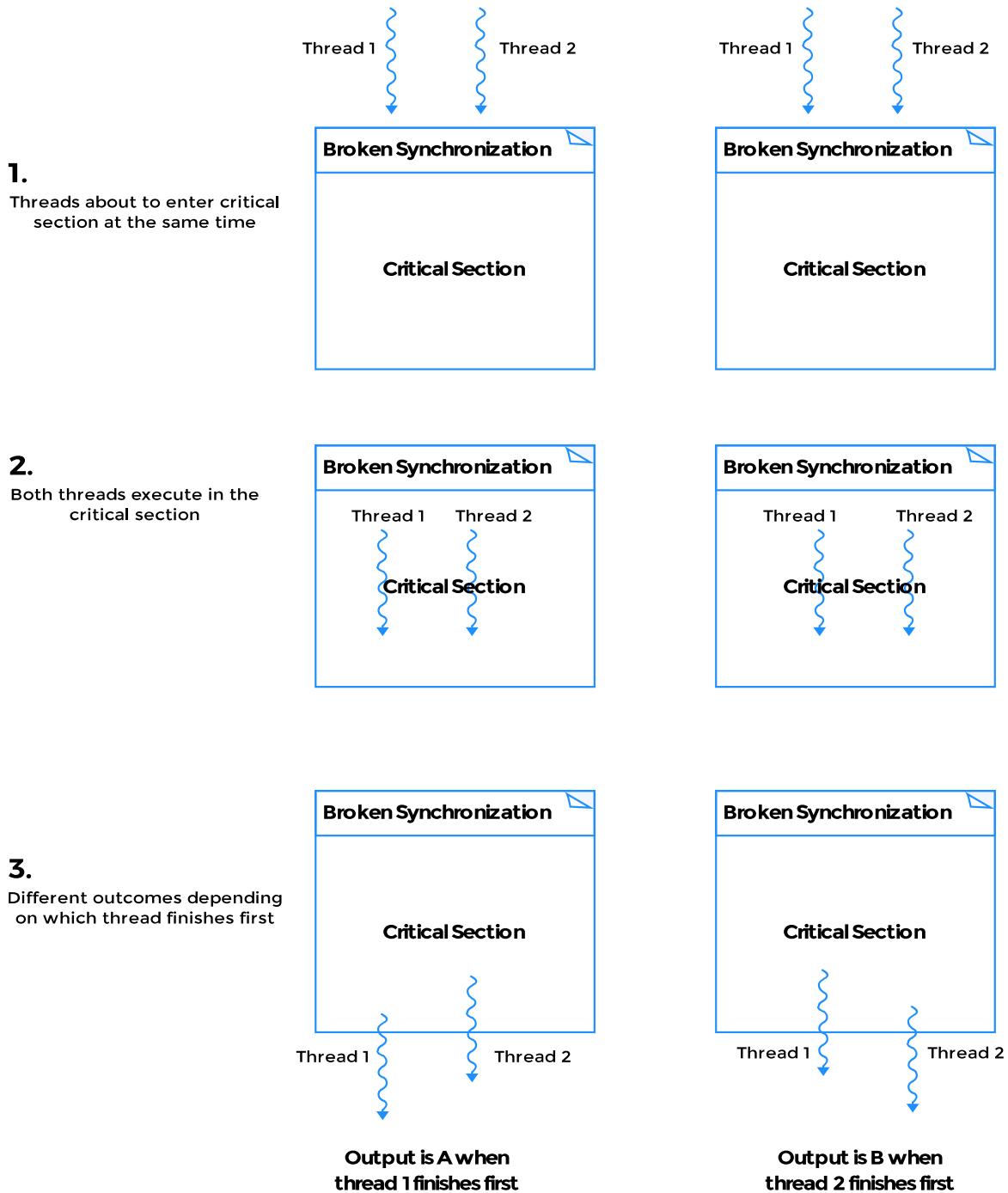
```
    Thread thread2 = new Thread(new Runnable() {
```

```
        @Override  
        public void run() {  
            rc.modifier();  
        }  
    });
```

```
    thread1.start();  
    thread2.start();
```

```
    thread1.join();  
    thread2.join();  
}
```

Below is a pictorial representation of what a race condition, in general, looks like.



Deadlocks, Liveness & Reentrant Locks

We discuss important concurrency concepts deadlock, liveness, live-lock, starvation and reentrant locks in depth. Also included are executable code examples for illustrating these concepts.

We'll cover the following

- o [DeadLock](#)
- o [Liveness](#)
- o [Live-Lock](#)
- o [Starvation](#)
- o [Deadlock Example](#)
- o [Reentrant Lock](#)

Logical follies committed in multithreaded code, while trying to avoid race conditions and guarding critical sections, can lead to a host of subtle and hard to find bugs and side-effects. Some of these incorrect usage patterns have their names and are discussed below.

DeadLock#

Deadlocks occur when two or more threads aren't able to make any progress because the resource required by the first thread is held by the second and the resource required by the second thread is held by the first.

Liveness#

Ability of a program or an application to execute in a timely manner is called liveness. If a program experiences a deadlock then it's not exhibiting liveness.

Live-Lock#

A live-lock occurs when two threads continuously react in response to the actions by the other thread without making any real progress. The best analogy is to think of two persons trying to cross each other in a hallway. John moves to the left to let Arun pass, and Arun moves to his right to let John pass. Both block each other now. John sees he's blocking Arun again and moves to his right and Arun moves to his left seeing he's blocking John. They never cross each other and keep blocking each other. This scenario is an example of a livelock. A process seems to be running and not deadlocked but in reality, isn't making any progress.

Starvation#

Other than a deadlock, an application thread can also experience starvation, when it never gets CPU time or access to shared resources. Other **greedy** threads continuously hog shared system resources not letting the starving thread make any progress.

Deadlock Example#

```
void increment(){

    acquire MUTEX_A
    acquire MUTEX_B
    // do work here
    release MUTEX_B
    release MUTEX_A

}

void decrement(){

    acquire MUTEX_B
    acquire MUTEX_A
    // do work here
    release MUTEX_A
    release MUTEX_B

}
```

The above code can potentially result in a deadlock. Note that deadlock may not always happen, but for certain execution sequences, deadlock can occur. Consider the below execution sequence that ends up in a deadlock:

```
T1 enters function increment
T1 acquires MUTEX_A
T1 gets context switched by the operating system
T2 enters function decrement
T2 acquires MUTEX_B
both threads are blocked now
```

Thread **T2** can't make progress as it requires **MUTEX_A** which is being held by **T1**. Now when **T1** wakes up, it can't make progress as it requires **MUTEX_B** and that is being held up by **T2**. This is a classic textbook example of a deadlock.

You can come back to the examples presented below as they require an understanding of the `synchronized` keyword that we cover in later sections. Or you can just run the examples and observe the output for now to get a high-level overview of the concepts we discussed in this lesson.

If you run the code snippet below, you'll see that the statements for acquiring locks: **lock1** and **lock2** print out but there's no progress after that and the execution times out. In this scenario, the deadlock occurs because the locks are being acquired in a nested fashion.

```
class Demonstration {  
  
    public static void main(String args[]) {  
        Deadlock deadlock = new Deadlock();  
        try {  
            deadlock.runTest();  
        } catch (InterruptedException ie) {  
        }  
    }  
}
```

```
class Deadlock {  
  
    private int counter = 0;  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
  
    Runnable incrementer = new Runnable() {  
  
        @Override  
        public void run() {  
            try {  
                for (int i = 0; i < 100; i++) {  
                    incrementCounter();  
                    System.out.println("Incrementing " + i);  
                }  
            } catch (InterruptedException ie) {  
            }  
        }  
    };  
}
```

```
Runnable decrementer = new Runnable() {  
  
    @Override  
    public void run() {  
        try {  
            for (int i = 0; i < 100; i++) {  
                decrementCounter();  
            }  
        } catch (InterruptedException ie) {  
        }  
    }  
};
```

```
        System.out.println("Decrementing " + i);
    }
} catch (InterruptedException ie) {
}

}

public void runTest() throws InterruptedException {

    Thread thread1 = new Thread(incrementer);
    Thread thread2 = new Thread(decrementer);

    thread1.start();
    // sleep to make sure thread 1 gets a chance to acquire lock1
    Thread.sleep(100);
    thread2.start();

    thread1.join();
    thread2.join();

    System.out.println("Done : " + counter);
}

void incrementCounter() throws InterruptedException {
    synchronized (lock1) {
        System.out.println("Acquired lock1");
        Thread.sleep(100);

        synchronized (lock2) {
            counter++;
        }
    }
}

void decrementCounter() throws InterruptedException {
    synchronized (lock2) {
        System.out.println("Acquired lock2");

        Thread.sleep(100);
        synchronized (lock1) {
            counter--;
        }
    }
}
```

Reentrant Lock#

Re-entrant locks allow for re-locking or re-entering of a synchronization lock. This can be best explained with an example. Consider the NonReentrant class below.

Take a minute to read the code and assure yourself that any object of this class if locked twice in succession would result in a deadlock. The same thread gets blocked on itself, and the program is unable to make any further progress. If you click run, the execution would time-out.

If a synchronization primitive doesn't allow reacquisition of itself by a thread that has already acquired it, then such a thread would block as soon as it attempts to reacquire the primitive a second time.

```
class Demonstration {  
  
    public static void main(String args[]) throws Exception {  
        NonReentrantLock nreLock = new NonReentrantLock();  
  
        // First locking would be successful  
        nreLock.lock();  
        System.out.println("Acquired first lock");  
  
        // Second locking results in a self deadlock  
        System.out.println("Trying to acquire second lock");  
        nreLock.lock();  
        System.out.println("Acquired second lock");  
    }  
}  
  
class NonReentrantLock {  
  
    boolean isLocked;  
  
    public NonReentrantLock() {  
        isLocked = false;  
    }  
  
    public synchronized void lock() throws InterruptedException {  
  
        while (isLocked) {  
            wait();  
        }  
        isLocked = true;  
    }  
}
```

```
public synchronized void unlock() {  
    isLocked = false;  
    notify();  
}  
}
```

The statement "Acquired second lock" is never printed

Mutex vs Semaphore

The concept of and the difference between a mutex and a semaphore will draw befuddled expressions on most developers' faces. We discuss the differences between the two most fundamental concurrency constructs offered by almost all language frameworks. Difference between a mutex and a semaphore makes a pet interview question for senior engineering positions!

We'll cover the following

- o [Mutex](#)
- o [Semaphore](#)
- o [Mutex Example](#)
- o [When a Semaphore Masquerades as a Mutex?](#)
- o [Semaphore for Signaling](#)
- o [Summary](#)

Having laid the foundation of concurrent programming concepts and their associated issues, we'll now discuss the all-important mechanisms of locking and signaling in multi-threaded applications and the differences amongst these constructs.

Mutex#

Mutex as the name hints implies mutual exclusion. A mutex is used to guard shared data such as a linked-list, an array or any primitive type. A mutex allows only a single thread to access a resource or critical section.

Once a thread acquires a mutex, all other threads attempting to acquire the same mutex are blocked until the first thread releases the mutex. Once released, most implementations arbitrarily chose one of the waiting threads to acquire the mutex and make progress.

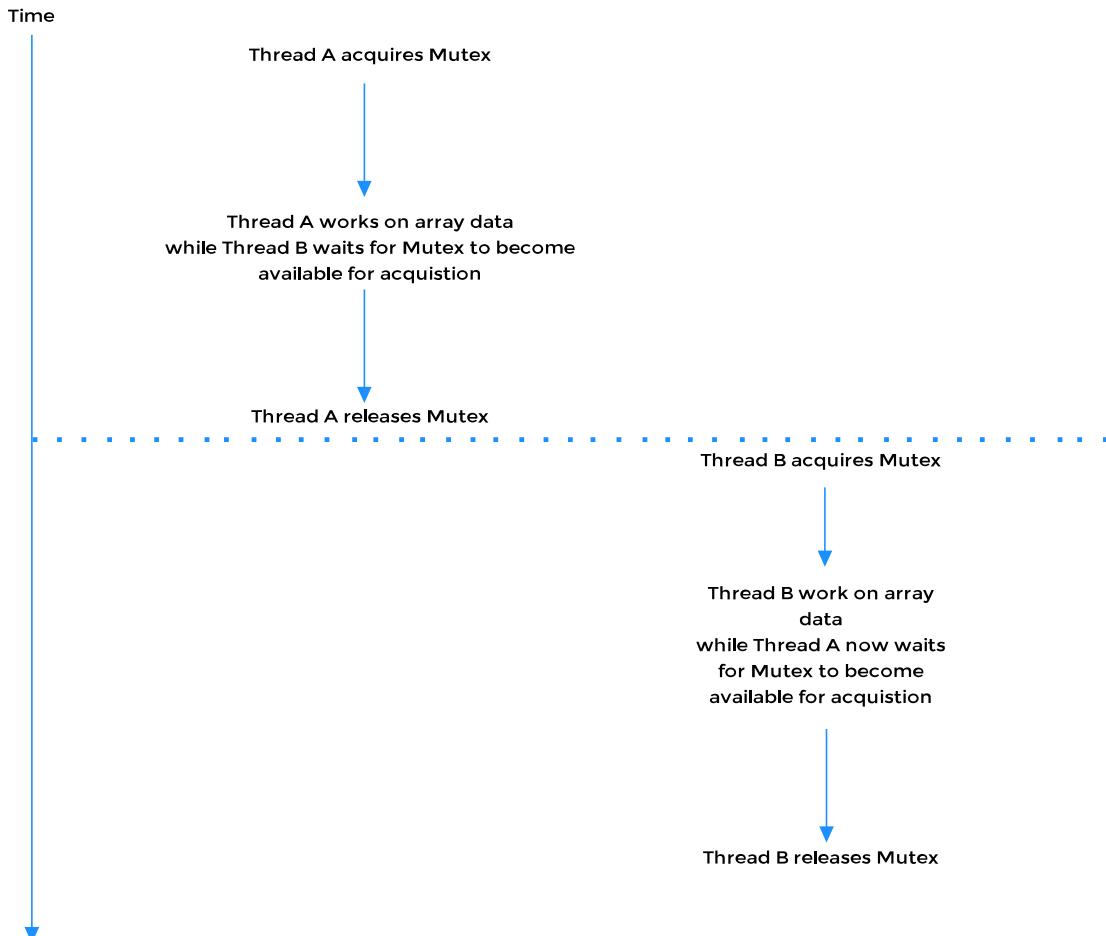
Semaphore#

Semaphore, on the other hand, is used for limiting access to a collection of resources. Think of semaphore as having a limited number of permits to give out. If a semaphore has given out all the permits it has, then any new thread that comes along requesting for a permit will be blocked, till an earlier thread with a permit returns it to the semaphore. A typical example would be a pool of database connections that can be handed out to requesting threads. Say there are ten available connections but 50 requesting threads. In such a scenario, a semaphore can only give out ten permits or connections at any given point in time.

A semaphore with a single permit is called a **binary semaphore** and is often thought of as an equivalent of a mutex, which isn't completely correct as we'll shortly explain. Semaphores can also be used for signaling among threads. This is an important distinction as it allows threads to cooperatively work towards completing a task. A mutex, on the other hand, is strictly limited to serializing access to shared state among competing threads.

Mutex Example#

The following illustration shows how two threads acquire and release a mutex one after the other to gain access to shared data. Mutex guarantees the shared state isn't corrupted when competing threads work on it.



When a Semaphore Masquerades as a Mutex?#

A semaphore can potentially act as a mutex if the permits it can give out is set to 1. However, the most important difference between the two is that in case of a mutex the same thread must call `acquire` and subsequent `release` on the mutex whereas in case of a binary semaphore, different threads can call acquire and release on the semaphore. The pthreads library documentation states this in the `pthread_mutex_unlock()` method's description.

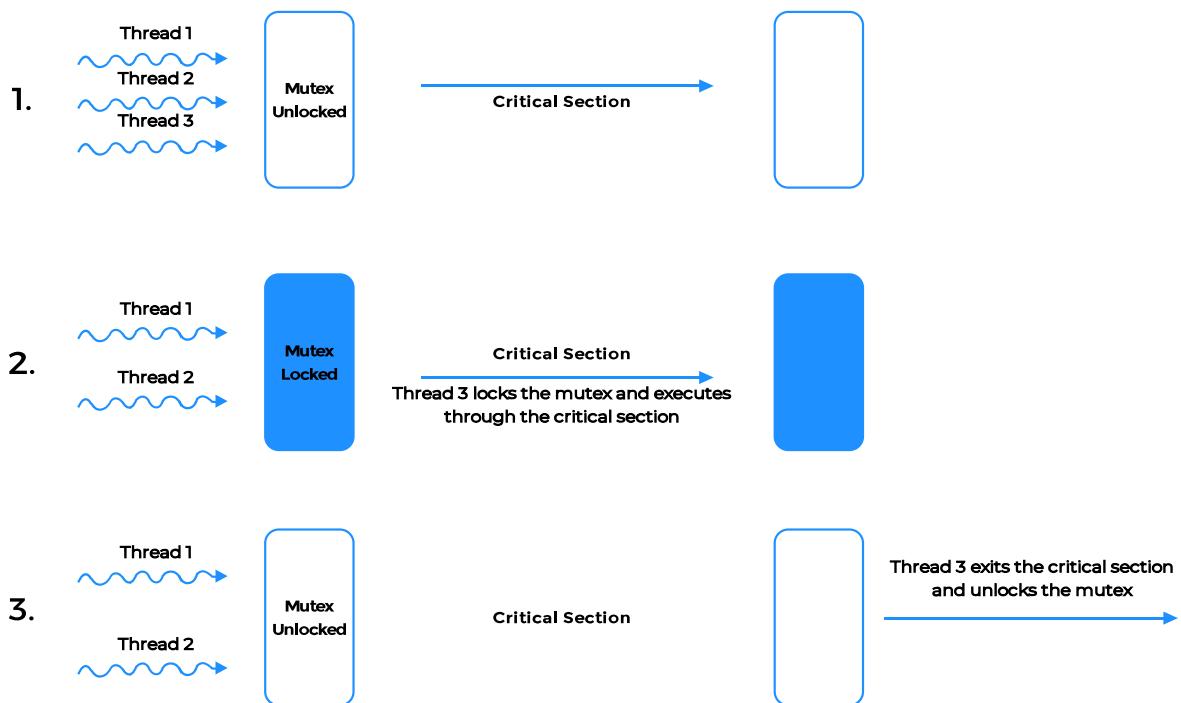
If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behavior results.

This leads us to the concept of **ownership**. A mutex is owned by the thread acquiring it till the point the owning-thread releases it, whereas for a semaphore there's no notion of ownership.

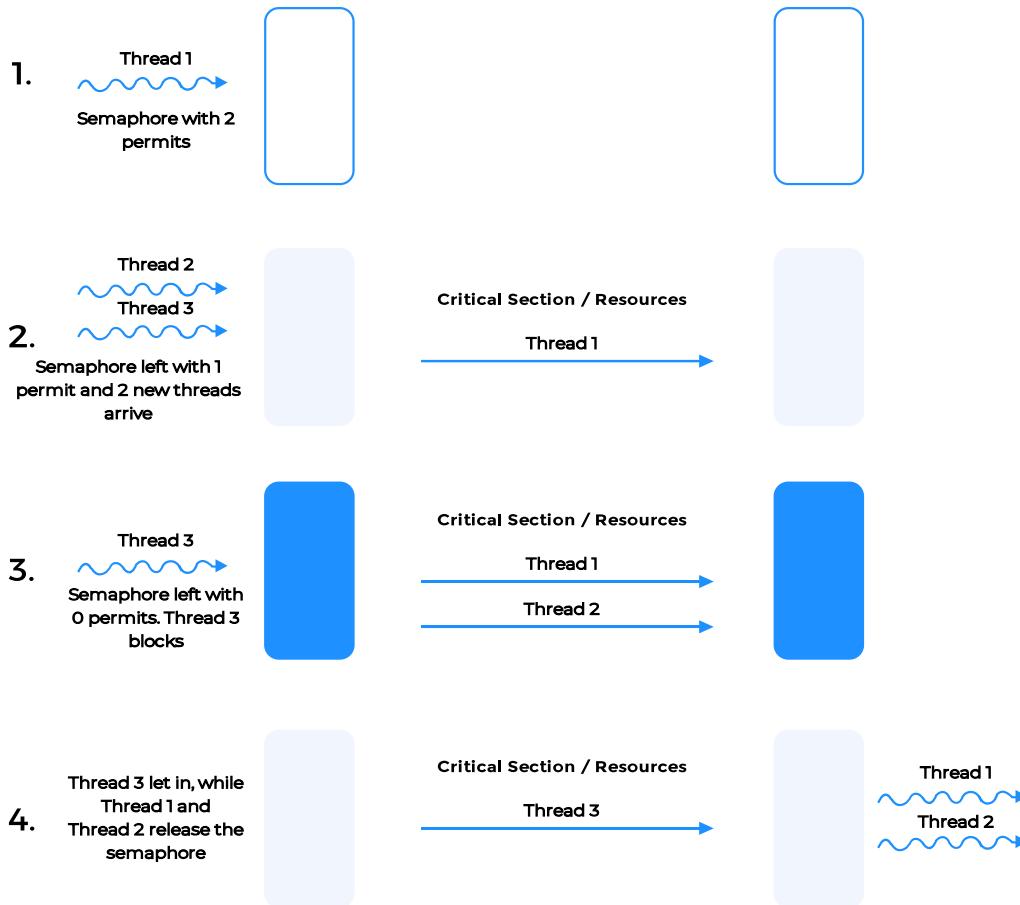
Semaphore for Signaling#

Another distinction between a semaphore and a mutex is that semaphores can be used for signaling amongst threads, for example in case of the classical **producer/consumer** problem the producer thread can signal the consumer thread by incrementing the semaphore count to indicate to the consumer thread to consume the freshly produced item. A mutex in contrast only guards access to shared data among competing threads by forcing threads to serialize their access to critical sections and shared data-structures.

Below is a pictorial representation of how a mutex works.



Below is a depiction of how a semaphore works. The semaphore initially has two permits and allows at most two threads to enter the critical section or access protected resources



Summary#

1. Mutex implies mutual exclusion and is used to serialize access to critical sections whereas semaphore can potentially be used as a mutex but it can also be used for cooperation and signaling amongst threads. Semaphore also solves the issue of missed signals.
2. Mutex is **owned** by a thread, whereas a semaphore has no concept of ownership.
3. Mutex if locked, must necessarily be unlocked by the same thread. A semaphore can be acted upon by different threads. This is true even if the semaphore has a permit of one
4. Think of semaphore analogous to a car rental service such as Hertz. Each outlet has a certain number of cars, it can rent out to customers. It can rent several cars to several customers at the same time but if all the cars are rented out then any new customers need to be put on a waitlist till one of the rented cars is returned. In contrast, think of a mutex like a lone runway on a remote airport. Only a single jet can

land or take-off from the runway at a given point in time. No other jet can use the runway simultaneously with the first aircraft.

Mutex vs Monitor

Learn what a monitor is and how it is different than a mutex. Monitors are advanced concurrency constructs and specific to languages frameworks.

We'll cover the following

- o [When Mutual Exclusion isn't Enough](#)
- o [Condition Variables](#)
- o [Why the while Loop](#)
- o [Monitor Explained](#)

Continuing our discussion from the previous section on locking and signaling mechanisms, we'll now pore over an advanced concept the **monitor**. It is exposed as a concurrency construct by some programming language frameworks including Java.

When Mutual Exclusion isn't Enough#

Concisely, a monitor is a mutex and then some. Monitors are generally language level constructs whereas mutex and semaphore are lower-level or OS provided constructs.

To understand monitors, let's first see the problem they solve. Usually, in multi-threaded applications, a thread needs to wait for some program predicate to be true before it can proceed forward. Think about a producer/consumer application. If the producer hasn't produced anything the consumer can't consume anything, so the consumer must **wait on** a predicate that lets the consumer know that something has indeed been produced. What could be a crude way of accomplishing this? The consumer could repeatedly check in a loop for the predicate to be set to true. The pattern would resemble the pseudocode below:

```
void busyWaitFunction() {  
    // acquire mutex  
    while (predicate is false) {  
        // release mutex  
        // acquire mutex  
    }  
    // do something useful
```

```
// release mutex  
}
```

Within the while loop we'll first release the mutex giving other threads a chance to acquire it and set the loop predicate to true. And before we check the loop predicate again, we make sure we have acquired the mutex again. This works but is an example of "spin waiting" which wastes a lot of CPU cycles. Next, let's see how condition variables solve the spin-waiting issue.

Condition Variables#

Mutex provides mutual exclusion, however, at times mutual exclusion is not enough. We want to test for a predicate with a mutually exclusive lock so that no other thread can change the predicate when we test for it but if we find the predicate to be false, we'd want to wait on a condition variable till the predicate's value is changed. This thus is the solution to spin waiting.

Conceptually, each condition variable exposes two methods `wait()` and `signal()`. The `wait()` method when called on the condition variable will cause the associated mutex to be atomically released, and the calling thread would be placed in a **wait queue**. There could already be other threads in the **wait queue** that previously invoked `wait()` on the condition variable. Since the mutex is now released, it gives other threads a chance to change the predicate that will eventually let the thread that was just placed in the wait queue to make progress. As an example, say we have a consumer thread that checks for the size of the buffer, finds it empty and invokes `wait()` on a condition variable. The predicate in this example would be **the size of the buffer**.

Now imagine a producer places an item in the buffer. The predicate, the size of the buffer, just changed and the producer wants to let the consumer threads know that there is an item to be consumed. This producer thread would then invoke `signal()` on the condition variable. The `signal()` method when called on a condition variable causes one of the threads that has been placed in the **wait queue** to get ready for execution. Note we didn't say the woken up thread starts executing, it just gets ready - and that could mean being placed in the ready queue. It is only after the producer thread which calls the `signal()` method has released the associated mutex that the thread in the ready queue starts executing. The thread in the ready queue must wait to acquire the mutex associated with the condition variable before it can start executing.

Lets see how this all translates into code.

```
void efficientWaitingFunction() {
```

```

    mutex.acquire()
    while (predicate == false) {
        condVar.wait()
    }
    // Do something useful
    mutex.release()
}

void changePredicate() {
    mutex.acquire()
    set predicate = true
    condVar.signal()
    mutex.release()
}

```

Let's dry run the above code. Say **thread A**

A executes `efficientWaitingFunction()` first and finds the loop predicate is false and enters the loop. Next **thread A** executes the statement `condVar.wait()` and is be placed in a wait queue. At the same time **thread A** gives up the mutex. Now **thread B** comes along and executes `changePredicate()` method. Since the mutex was given up by **thread A**, **thread B** is be able to acquire it and set the predicate to true. Next it signals the condition variable `condVar.signal()`. This step places **thread A** into the ready queue but **thread A** doesn't start executing until **thread B** has released the mutex.

Note that the order of signaling the condition variable and releasing the mutex can be interchanged, but generally, the preference is to signal first and then release the mutex. However, the ordering might have ramifications on thread scheduling depending on the threading implementation.

Why the `while` Loop#

The wary reader would have noticed us using a while loop to test for the predicate. After all, the pseudocode could have been written as follows

```

void efficientWaitingFunction() {
    mutex.acquire()
    if (predicate == false) {
        condVar.wait()
    }
    // Do something useful
    mutex.release()
}

```

If the snippet is re-written in the above manner using an `if` clause instead of a `while` then,, we need a guarantee that once the variable `condVar` is

signaled, the predicate can't be changed by any other thread and that the signaled thread becomes the owner of the monitor. This may not be true. For one, a different thread could get scheduled and change the predicate back to false before the signaled thread gets a chance to execute, therefore the signaled thread must check the predicate again, once it acquires the monitor. Secondly, use of the loop is necessitated by design choices of monitors that we'll explore in the next section. Last but not the least, on POSIX systems, **spurious or fake wakeups** are possible (also discussed in later chapters) even though the condition variable has not been signaled and the predicate hasn't changed. The idiomatic and correct usage of a monitor dictates that the predicate always be tested for in a while loop.

Monitor Explained#

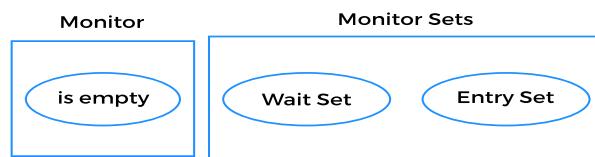
After the above discussion, we can now realize that a monitor is made up of a mutex and one or more condition variables. A single monitor can have multiple condition variables but not vice versa. Theoretically, another way to think about a monitor is to consider it as an entity having two queues or sets where threads can be placed. One is the **entry set** and the other is the **wait set**. When a thread A **enters** a monitor it is placed into the **entry set**. If no other thread **owns** the monitor, which is equivalent of saying no thread is actively executing within the monitor section, then thread A will **acquire** the monitor and is said to own it too. Thread A will continue to execute within the monitor section till it **exits** the monitor or calls `wait()` on an associated condition variable and be placed into the wait set. While thread A **owns** the monitor no other thread will be able to execute any of the critical sections protected by the monitor. New threads requesting ownership of the monitor get placed into the **entry set**.

Continuing with our hypothetical example, say another thread B comes along and gets placed in the **entry set**, while thread A sits in the **wait set**. Since no other thread owns the monitor, thread B successfully acquires the monitor and continues execution. If thread B exits the monitor section without calling `notify()` on the condition variable, then thread A will remain waiting in the **wait set**. Thread B can also invoke `wait()` and be placed in the **wait set** along with thread A. This then would require a third thread to come along and call `notify()` on the condition variable on which both threads A and B are waiting. Note that only a single thread will be able to **own** the monitor at any given point in time and will have exclusive access to data structures or critical sections protected by the monitor.

Practically, in Java each object is a monitor and implicitly has a lock and is a condition variable too. You can think of a monitor as a mutex with a wait set. Monitors allow threads to exercise **mutual exclusion** as well as **cooperation** by allowing them to wait and signal on conditions.

A pictorial representation of the above simulation appears below:

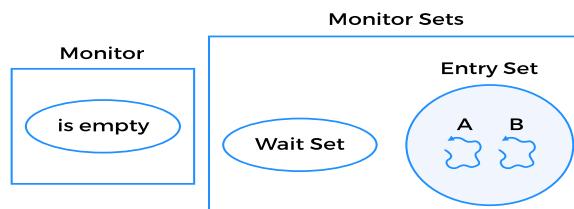
1. Initial Monitor State



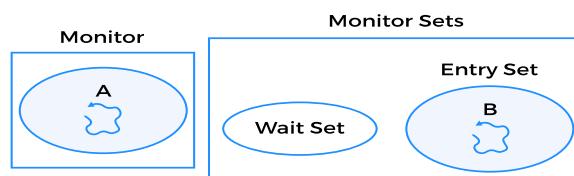
2. Two threads come along to enter the monitor



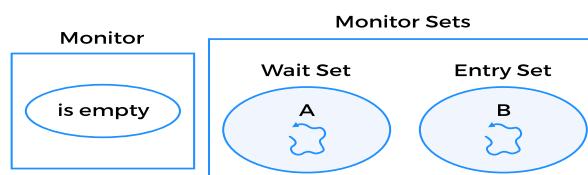
3. Thread A and B get placed in the entry set



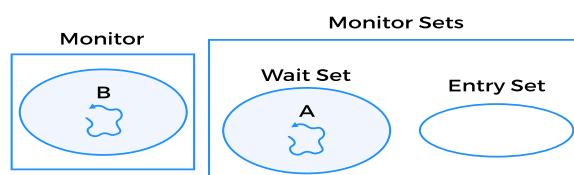
4. Thread A enters the monitor and starts execution



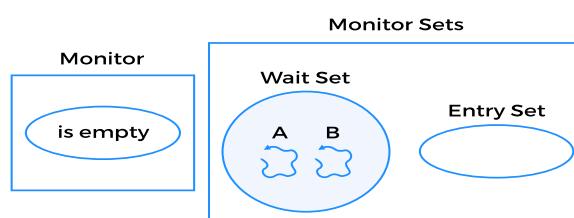
5. Thread A execution Wait() and gets placed in wait set



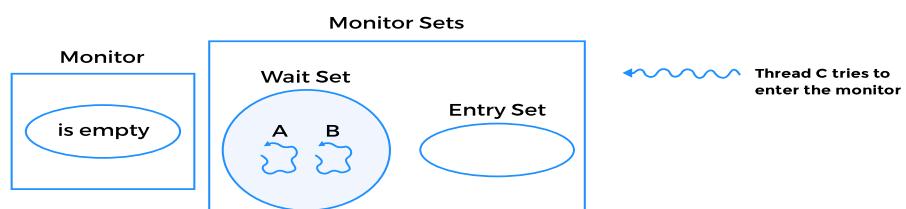
6. Thread B now able to enter the monitor



7. Thread B also invokes wait() and gets placed in the wait set



8. Thread C comes along to enter the monitor



Java's Monitor & Hoare vs Mesa Monitors

Continues the discussion of the differences between a mutex and a monitor and also looks at Java's implementation of the monitor.

We'll cover the following

- [Java's Monitor](#)
- [Bad Synchronization Example 1](#)
- [Bad Synchronization Example 2](#)
- [Hoare vs Mesa Monitors](#)

We discussed the abstract concept of a monitor in the previous section and now let's see the working of a concrete implementation of it in Java.

Java's Monitor#

In Java every object is a condition variable and has an associated lock that is hidden from the developer. Each java object exposes `wait()` and `notify()` methods.

Before we execute `wait()` on a java object we need to lock its hidden mutex. That is done implicitly through the **synchronized** keyword. If you attempt to call `wait()` or `notify()` outside of a synchronized block, an **IllegalMonitorStateException** would occur. It's Java reminding the developer that the mutex wasn't acquired before wait on the condition variable was invoked. `wait()` and `notify()` can only be called on an object once the calling thread becomes the **owner** of the monitor. The ownership of the monitor can be achieved in the following ways:

- the method the thread is executing has synchronized in its signature
- the thread is executing a block that is synchronized on the object on which wait or notify will be called
- in case of a class, the thread is executing a static method which is synchronized.

Bad Synchronization Example 1#

In the below snippet, `wait()` is being called outside of a synchronized block, i.e. without implicitly locking the associated mutex. Running the code results in **IllegalMonitorStateException**

```

class BadSynchronization {

    public static void main(String args[]) throws InterruptedException {
        Object dummyObject = new Object();

        // Attempting to call wait() on the object
        // outside of a synchronized block.
        dummyObject.wait();
    }
}

```

Bad Synchronization Example 2#

Here's another example where we try to call `notify()` on an object in a synchronized block which is synchronized on a different object. Running the code will again result in an **IllegalMonitorStateException**

```

class BadSynchronization {

    public static void main(String args[]) {
        Object dummyObject = new Object();
        Object lock = new Object();

        synchronized (lock) {
            lock.notify();

            // Attempting to call notify() on the object
            // in synchronized block of another object
            dummyObject.notify();
        }
    }
}

```

Hoare vs Mesa Monitors#

So far we have determined that the idiomatic usage of a monitor requires using a while loop as follows. Let's see how the design of monitors affects this recommendation.

```

while( condition == false ) {
    condVar.wait();
}

```

Once the asleep thread is signaled and wakes up, you may ask why does it need to check for the condition being false again, the signaling thread must have just set the condition to true?

In **Mesa monitors** - Mesa being a language developed by Xerox researchers in the 1970s - it is possible that the time gap between thread B calls `notify()` and releases its mutex **and** the instant at which the asleep thread A, wakes up and reacquires the mutex, ***the predicate is changed back to false by another thread different than the signaler and the awoken threads!*** The woken up thread competes with other threads to acquire the mutex once the signaling thread B **empties** the monitor. On signaling, thread B doesn't give up the monitor just yet; rather it continues to own the monitor until it exits the monitor section.

In contrast, **Hoare monitors** - Hoare being one of the original inventor of monitors - the signaling thread B **yields** the monitor to the woken up thread A and thread A **enters** the monitor, while thread B sits out. This guarantees that the predicate will not have changed and instead of checking for the predicate in a while loop an if-clause would suffice. The woken-up/released thread A immediately starts execution when the signaling thread B signals that the predicate has changed. No other thread gets a chance to change the predicate since no other thread gets to enter the monitor.

Java, in particular, subscribes to Mesa monitor semantics and the developer is always expected to check for condition/predicate in a while loop. Mesa monitors are more efficient than Hoare monitors.

Semaphore vs Monitor

This lesson discusses the differences between a monitor and a semaphore.

We'll cover the following

- o [Difference between Semaphore and Monitor](#)

Monitor, mutex, and semaphores can be confusing concepts initially. A **monitor is made up of a mutex and a condition variable**. One can think of a mutex as a subset of a monitor. Differences between a monitor and semaphore are discussed below.

Difference between Semaphore and Monitor#

- A monitor and a semaphore are interchangeable and theoretically, one can be constructed out of the other or one can be reduced to the other. However, monitors take care of atomically acquiring the necessary locks whereas, with semaphores, the onus of appropriately acquiring and releasing locks is on the developer, which can be error-prone.
- Semaphores are lightweight when compared to monitors, which are bloated. However, the tendency to misuse semaphores is far greater than monitors. When using a semaphore and mutex pair as an alternative to a monitor, it is easy to lock the wrong mutex or just forget to lock altogether. Even though both constructs can be used to solve the same problem, monitors provide a pre-packaged solution with less dependency on a developer's skill to get the locking right.
- Java monitors enforce correct locking by throwing the `IllegalMonitorStateException` exception object when methods on a condition variable are invoked without first acquiring the associated lock. The exception is in a way saying that either the object's lock/mutex was not acquired at all or that an incorrect lock was acquired.
- A semaphore can allow several threads access to a given resource or critical section, however, only a single thread at any point in time can own the monitor and access associated resource.
- Semaphores can be used to address the issue of **missed signals**, however with monitors additional state, called the predicate, needs to be maintained apart from the condition variable and the mutex which make up the monitor, to solve the issue of missed signals.

Amdahl's Law

Blindly adding threads to speed up program execution may not always be a good idea. Find out what Amdahl's Law says about parallelizing a program

We'll cover the following

- [Definition](#)
- [Example](#)

[Definition#](#)

No text on concurrency is complete without mentioning the ***Amdahl's Law***. The law specifies the cap on the maximum speedup that can be achieved when parallelizing the execution of a program.

If you have a poultry farm where a hundred hens lay eggs each day, then no matter how many people you hire to process the laid eggs, you still need to wait an entire day for the 100 eggs to be laid. Increasing the number of workers on the farm can't shorten the time it takes for a hen to lay an egg. Similarly, software programs consist of parts which can't be sped up even if the number of processors is increased. These parts of the program must execute serially and aren't amenable to parallelism.

Amdahl's law describes the theoretical speedup a program can achieve at best by using additional computing resources. We'll skip the mathematical derivation and go straight to the simplified equation expressing Amdahl's law:

$$S(n) = \frac{1}{(1 - P) + \frac{P}{n}}$$

- **$S(n)$** is the speed-up achieved by using n cores or threads.
- P is the fraction of the program that is parallelizable
- $(1 - P)$ is the fraction of the program that must be executed serially.

Example#

Say our program has a parallelizable portion of $P = 90\% = 0.9$. Now let's see how the speed-up occurs as we increase the number of processes

- **$n = 1$ processor**

$$S(1) = \frac{1}{(1 - P) + \frac{P}{1}} = \frac{1}{1 - P + P} = 1$$

- **$n = 2$ processors**

$$S(2) = \frac{1}{(1 - 0.9) + \frac{0.9}{2}} = \frac{1}{0.1 + 0.45} = 1.81$$

- **$n = 5$ processors**

$$S(5) = \frac{1}{(1 - 0.9) + \frac{0.9}{5}} = \frac{1}{0.1 + 0.18} = 3.57$$

- **$n = 10$ processors**

$$S(10) = \frac{1}{(1 - 0.9) + \frac{0.9}{10}} = \frac{1}{0.1 + 0.09} = 5.26$$

- **$n = 100$ processors**

$$S(100) = \frac{1}{(1 - 0.9) + \frac{0.9}{100}} = \frac{1}{0.1 + 0.009} = 9.17$$

- **$n = 1000$ processors**

$$S(1000) = \frac{1}{(1 - 0.9) + \frac{0.9}{1000}} = \frac{1}{0.1 + 0.0009} = 9.91$$

- **$n = infinite$ processors**

$$S(\infty) = \frac{1}{(1 - 0.9) + \frac{0.9}{\infty}} = \frac{1}{0.1 + 0} = 10$$

The speed-up steadily increases as we increase the number of processors or threads. However, as you can see the theoretical maximum speed-up for our program with 10% serial execution will be 10. We can't speed-up our program execution more than 10 times compared to when we run the same program on a single CPU or thread. To achieve greater speed-ups than 10 we must optimize or parallelize the serially executed portion of the code.

Another important aspect to realize is that when we speed-up our program execution by roughly 5 times, we do so by employing 10 processors. The utilization of these 10 processors, in turn, decreases by roughly 50% because now the 10 processors remain idle for the rest of the time that a single processor would have been busy. Utilization is defined as the ***speedup divided by the number of processors***.

As an example say the program runs in 10 minutes using a single core. We assumed the parallelizable portion of the program is 90%, which implies 1 minute of the program time must execute serially. The speedup we can

achieve with 10 processors is roughly 5 times which comes out to be 2 minutes of total program execution time. Out of those 2 minutes, 1 minute is of mandatory serial execution and the rest can all be parallelized. This implies that 9 of the processors will complete 90% of the non-serial work in 1 minute while 1 processor remains idle and then one out of the 10 processors, will execute the serial portion for another minute. The rest of the 9 processors are idle for that 1 minute during which the serial execution takes place. In effect, the combined utilization of the ten processors drops by 50%.

As N approaches infinity, the Amdahl's law takes the following form:

$$S(n) = \frac{1}{(1 - P)} = \frac{1}{\textit{fraction of program serially executed}}$$

One should take the calculations using Amdahl's law with a grain of salt. If the formula spits out a speed-up of 5x it doesn't imply that in reality one would observe a similar speed-up. There are other factors such as the memory architecture, cache misses, network and disk I/O etc that can affect the execution time of a program and the actual speed-up might be less than the calculated one.

The Amdahl's law works on a problem of fixed size. However as computing resources are improved, algorithms run on larger and even larger datasets. As the dataset size grows, the parallelizable portion of the program grows faster than the serial portion and a more realistic assessment of performance is given by **Gustafson's law**, which we won't discuss here as it is beyond the scope of this text.

Moore's Law

Discusses impact of Moore's law on concurrency.

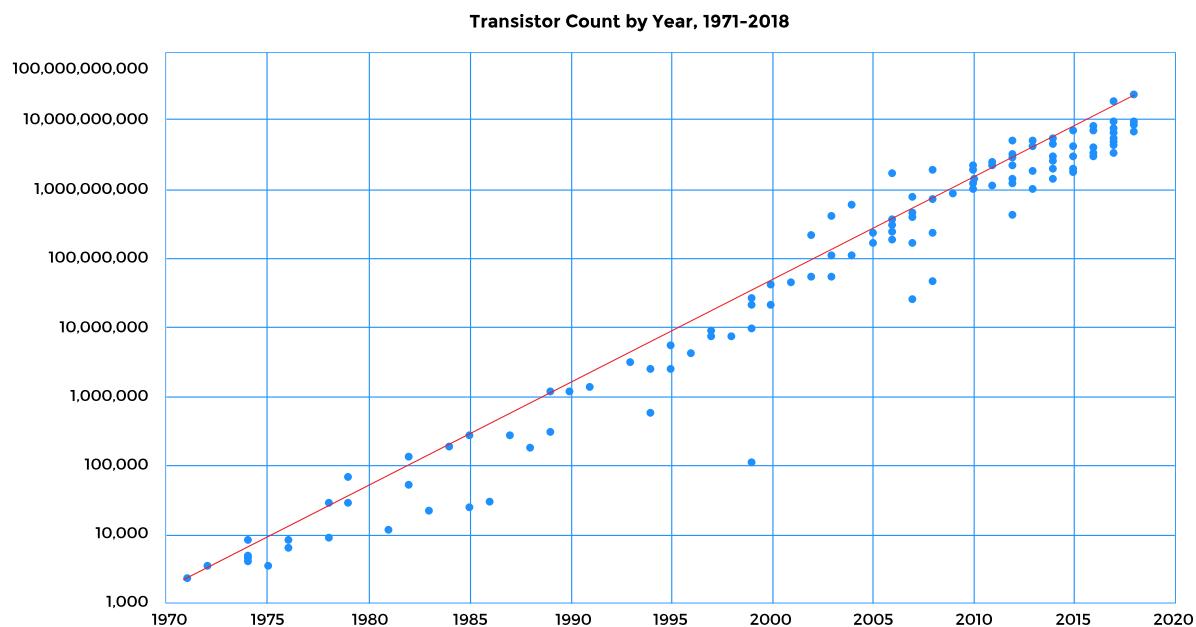
We'll cover the following

- Moore's Law

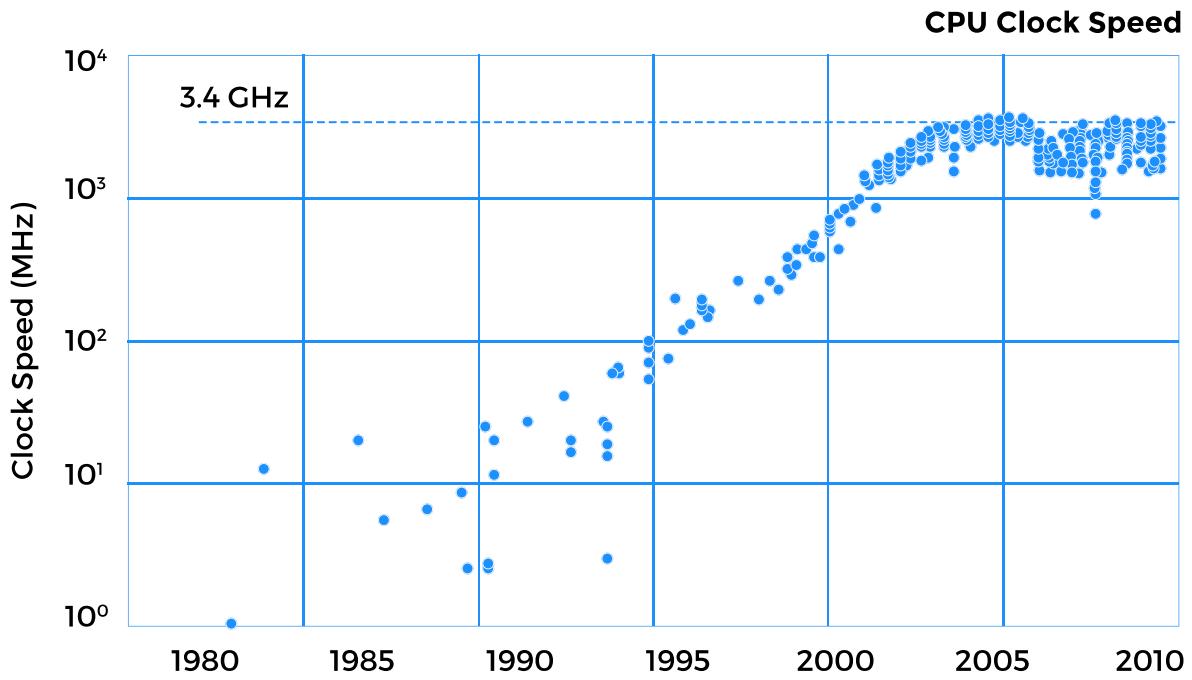
Moore's Law#

In this lesson, we'll cover a high-level overview of Moore's law to present a final motivation to the reader for writing multi-threaded applications and realize that concurrency is inevitable in the future of software.

[Gordon Moore](#), co-founder of Intel, observed the number of transistors that can be packed into a given unit of space doubles about every two years and in turn the processing power of computers doubles and the cost halves. Moore's law is more of an observation than a law grounded in formal scientific research. It states that **the number of transistors per square inch on a chip will double every two years**. This exponential growth has been going on since the 70's and is only now starting to slow down. The following graph shows the growth of the transistor count.



Initially, the clock speeds of processors also doubled along with the transistor count. This is because as transistors get smaller, their frequency increases and propagation delays decrease because now the transistors are packed closer together. However, the promise of exponential growth by Moore's law came to an end more than a decade ago with respect to clock speeds. The increase in clock speeds of processors has slowed down much faster than the increase in number of transistors that can be placed on a microchip. If we plot clock speeds we find that the linear exponential growth stopped after 2003 and the trend line flattened out. The clock speed (proportional to difference between supply voltage and threshold voltage) cannot increase because the supply voltage is already down to an extent where it cannot be decreased to get dramatic gains in clock speed. **In 10 years from 2000 to 2009, clock speed just increased from 1.3 GHz to 2.8 GHz merely doubling in a decade rather than increasing 32 times as expected by Moore's law.** The following plot shows the clock speeds flattening out towards 2010.



Since processors aren't getting faster as quickly as they used to, we need alternative measures to achieve performance gains. One of the ways to do that is to use multicore processors. Introduced in the early 2000s, multicore processors have more than one CPU on the same machine. To exploit this processing power, programs must be written as multi-threaded applications. A single-threaded application running on an octa-core processor will only use 1/8th of the total throughput of that machine, which is unacceptable in most scenarios.

Another analogy is to think of a bullock cart being pulled by an ox. We can breed the ox to be stronger and more powerful to pull more load but eventually there's a limit to how strong the ox can get. To pull more load, an easier solution is to attach several oxen to the bullock cart. The computing industry is also going in the direction of this analogy.

Atomic Assignments

Not all assignment operations in Java are atomic. Learn what types can be atomically assigned according to the Java specification.

We'll cover the following

- [Overview](#)
- [Java specification](#)

Overview#

Multithreading revolves around the ability to execute actions atomically, that is without interference from other threads. We use various language provided constructs to ensure that critical sections of code are executed atomically. However, this also begs the question, what operations are performed atomically by the language. For instance we already know that incrementing an integer counter as follows is never thread-safe:

```
counter++;
```

The above expression breaks down into several lower-level instructions that may or may not be performed atomically and therefore we can't guarantee that the execution of this expression is thread-safe. However, what about simple operations such as an assignment? Consider the following:

```
void someMethod(int passedInt) {  
    counter = passedInt; // counter is an instance variable  
}
```

If several threads were to invoke `someMethod` and each passing a different value for the integer can we assume the assignment will be thread-safe? To make the example concrete, say we have two threads one invoking `someMethod(5)` and the other invoking `someMethod(7)`. There are three outcomes for the counter

`counter` is assigned 5 `counter` is assigned 7 `counter` has an arbitrary value because some bits are written by the first thread and some by the second thread.

In our example the variable `counter` is of primitive type `int` which is 32 bits in Java. The astute reader would question what guarantees an assignment operation as atomic?

Java specification#

The question if assignment is atomic or not is a valid one and the confusion is addressed by the Java specification itself, which states:

- Assignments and reads for primitive data types except for `double` and `long` are always atomic. If two threads are invoking `someMethod()` and passing in 5 and 7 for the integer `counter` variable then the variable will hold either 5 or 7 and not any other value. There will be no partial writes of bits from either thread.

- **The reads and writes to `double` and `long` primitive types aren't atomic.**
The JVM specification allows implementations to break-up the write of the 64 bit `double` or `long` primitive type into two writes, one for each 32 bit half. This can result in a situation where two threads are simultaneously writing a `double` or `long` value and a third thread observes the first 32 bits from the write by the first thread and the next 32 bits from the write by the second thread. As a result the third thread reads a value that has neither been written by either of the two threads or is a garbage value. In order to make reads and writes to `double` or `long` primitive types atomic, we must mark them as `volatile`. The specification guarantees writes and reads to `volatile double` and `long` primitive types as atomic. Note that some JVM implementations may make the writes and reads to `double` and `long` types as atomic but this isn't universally guaranteed across all the JVM implementations. We'll cover `volatile` in a later lesson, for now, just remember that primitive `double` and `long` types must be marked as `volatile` for atomic assignments.
- All reference assignments are atomic. By reference we mean a variable holding a memory location address, where an object has been allocated by the JVM. For instance, consider the snippet `Thread currentThread = Thread.currentThread();` The variable `currentThread` holds the address for the current thread's object. If several threads execute the above snippet, the variable `currentThread` will hold one valid memory location address and not a garbage value. It can't happen that the variable `currentThread` holds some bytes from the assignment operation of one thread and other bytes from the assignment operation of an another thread. Whatever reference the variable holds will reflect an assignment from one of the threads. Reference reads and writes are always atomic whether the reference (memory address) itself consists of 32 or 64 bits.

Bear in mind that atomic assignments promised by the Java platform don't imply thread-safety! Our example method `someMethod()` is not thread-safe. Consider more complex situations such as reading and then assigning a variable in a method (e.g. initializing a singleton), such scenarios need to be made thread-safe using locks or `synchronized`.

Thready Safety & Synchronized

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

We'll cover the following

- Thread Safe
- Synchronized

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 sleep");
            Thread.sleep(5000);
            System.out.println("Woke up and about to invoke wait()");
            flag.wait();
        }
    } catch (InterruptedException ie) {

    }
}

Thread t2 = new Thread(new Runnable() {

    public void run() {
        flag = false;
        System.out.println("Boolean assignment done.");
    }
});

t1.start();
Thread.sleep(1000);
t2.start();
t1.join();
t2.join();
}

public static void runExample() throws InterruptedException {
    IncorrectSynchronization incorrectSynchronization = new IncorrectSynchronization();
    incorrectSynchronization.example();
}
}

```

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.

Wait & Notify

We'll cover the following

- o **wait()**

- o `notify()`
- o `notifyAll()`

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.

Interrupting Threads

We'll cover the following

- o `InterruptedException`

InterruptedException#

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
for an hour.");
                    Thread.sleep(1000 * 60 * 60);
                } catch (InterruptedException ie) {
                    System.out.println("The interrupt flag is cleared :
" + Thread.interrupted() + " " +
Thread.currentThread().isInterrupted());
                    Thread.currentThread().interrupt();
                    System.out.println("Oh someone woke me up ! ");
                    System.out.println("The interrupt flag is set now :
" + Thread.currentThread().isInterrupted() + " " +
Thread.interrupted());

                }
            }
        });
    }
}
```

```

        sleepyThread.start();

        System.out.println("About to wake up the sleepy thread ...");
        sleepyThread.interrupt();
        System.out.println("Woke up sleepy thread ...");

        sleepyThread.join();
    }
}

```

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*.

Volatile

We'll cover the following

- [Explanation](#)
- [Example](#)
- [When is volatile thread-safe](#)

Explanation#

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.

Example#

Consider the program below:

```
public class VolatileExample {  
    boolean flag = false;  
  
    void threadA() {  
        while (!flag) {  
            // ... Do something useful  
        }  
    }  
  
    void threadB() {  
        flag = true;  
    }  
}
```

In the above program, we would expect that `threadA` would exit the `while` loop once `threadB` sets the variable `flag` to true but `threadA` may unfortunately find itself spinning forever if it has cached the variable `flag`'s value. In this scenario, marking `flag` as `volatile` will fix the problem. Note that `volatile` presents a consistent view of the memory to all the threads. However, remember that `volatile` doesn't imply or mean thread-

safety. Consider the program below where we declare the variable `count volatile` and several threads increment the variable a 1000 times each. If you run the program several times you'll see `count` summing up to values other than the expected 10,000.

```
class Demonstration {  
  
    // volatile doesn't imply thread-safety!  
    static volatile int count = 0;  
  
    public static void main(String[] args) throws InterruptedException {  
  
        int numThreads = 10;  
        Thread[] threads = new Thread[numThreads];  
  
        for (int i = 0; i < numThreads; i++) {  
            threads[i] = new Thread(new Runnable() {  
                @Override  
                public void run() {  
                    for (int i = 0; i < 1000; i++)  
                        count++;  
                }  
            });  
        }  
  
        for (int i = 0; i < numThreads; i++) {  
            threads[i].start();  
        }  
  
        for (int i = 0; i < numThreads; i++) {  
            threads[i].join();  
        }  
  
        System.out.println("count = " + count);  
    }  
}
```

When is **volatile** thread-safe#

Volatile comes into play because of multiple 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.**

Reentrant Locks & Condition Variables

We'll cover the following

- o Reentrant Lock
- o Condition Variable
- o `java.util.concurrent`

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 we can 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](#).

Missed Signals

We'll cover the following

- 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.

```
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() {

    public void run() {

        lock.lock();

        try {
            condition.await();
            System.out.println("Received signal");
        } catch (InterruptedException ie) {
            // handle interruption
        }

        lock.unlock();
    }
});

signaller.start();
signaller.join();

waiter.start();
waiter.join();

System.out.println("Program Exiting.");
}
}

```

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() {

            public void run() {
                try {
                    semaphore.acquire();
                    System.out.println("Received signal");
                } catch (InterruptedException ie) {
                    // handle interruption
                }
            }
        });

        signaller.start();
        signaller.join();
        Thread.sleep(5000);
        waiter.start();
        waiter.join();

        System.out.println("Program Exiting.");
    }
}

```

Semaphore in Java

We'll cover the following

- [Explanation](#)

Explanation#

Java's semaphore can be `release()`-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.

Incorrect Use of Semaphore

```
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
                    // exception that causes the thread to crash.
                    throw new RuntimeException("exception happens at runtime.");

                    // The following line to signal the semaphore is never reached
                    // semaphore.release();
                }
            }
        });

        badThread.start();

        // Wait for the bad thread to go belly-up
    }
}
```

```

Thread.sleep(1000);

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

    public void run() {
        System.out.println("Good thread patiently waiting to be signalled.");
        try {
            semaphore.acquire();
        } catch (InterruptedException ie) {
            // handle thread interruption
        }
    }
});

goodThread.start();
badThread.join();
goodThread.join();
System.out.println("Exiting Program");
}
}

```

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.

```
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 the function
    return;
} finally {
    System.out.println("Bad thread releasing semaphore.");
    semaphore.release();
}

} catch (InterruptedException ie) {
    // handle thread interruption
}
}
});
};

badThread.start();

// Wait for the bad thread to go belly-up
Thread.sleep(1000);

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

    public void run() {
        System.out.println("Good thread patiently waiting to be signalled.");
        try {
            semaphore.acquire();
        } catch (InterruptedException ie) {
            // handle thread interruption
        }
    }
});

goodThread.start();
badThread.join();
goodThread.join();
System.out.println("Exiting Program");
}
}

```

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

Spurious Wakeups

We'll cover the following

- o Waking-up for no reason

Waking-up for no reason#

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, interrupted, or
* timing out, a so-called <i>spurious wakeup</i>. While this will rarely
* occur in practice, applications must guard against it by testing for
* the condition that should have caused the thread to be awakened and
* continuing to wait if the condition is not satisfied. In 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
*     }
*
```

Atomic Classes

Understand the performance benefits of atomic classes over locks.

We'll cover the following

- [Introduction](#)
- [Cons of Locking](#)
 - [Thread scheduling vs useful work](#)
 - [Priority inversion](#)
 - [Liveness issues](#)
 - [Locking, a heavyweight mechanism](#)
- [Atomic vs volatile](#)
- [Atomic Processor Instructions](#)
 - [Compare and Swap](#)
 - [ABA Problem](#)

Introduction#

As part of your senior engineer interview, you may be quizzed on the atomic classes and their utility. Though this is an advanced topic, understanding of it broadens your depth about, and command over concurrent systems and their workings.

So far we have seen locks that allow shared data to be manipulated safely among multiple threads. However, locking doesn't come for free and takes a toll on performance especially when the shared data/state is being contented for access by multiple threads.

Cons of Locking#

Locking comes with its downsides some of which include:

Thread scheduling vs useful work#

JVM is very efficient when it comes to acquiring and releasing a lock that is requested by a single thread. However, when multiple threads attempt to acquire the same lock, only one wins and the rest must be suspended. The suspension and resumption of threads is costly and introduces significant overhead and this can be an issue for scenarios where several threads contend for the same lock but execute very little functionality. In such cases, the time spent in scheduling overhead overwhelms the useful work done. This is true of synchronized collections where the majority of methods perform very few operations.

Priority inversion#

A higher priority thread can be blocked by a lower priority thread that holds the lock and itself is blocked because of a page fault, scheduling delay etc. This situation effectively downgrades the priority of the higher-priority thread to that of the lower-priority thread since the former can't make progress until the latter releases the lock. In general, all threads that require a particular lock can't make progress until the thread holding the lock releases it.

Liveness issues#

Locking also introduces the possibility of liveness issues such as deadlocks, livelock or simply programming bugs that have threads caught in infinite loops blocking other threads from making progress.

Locking, a heavyweight mechanism#

In general locking is a heavyweight mechanism, especially for fine-grained tasks such as manipulating a counter. Locking is akin to assuming the worst or preparing for the worst possible scenario, i.e. the thread assumes it would necessarily run into contention with another thread and acquires a lock to manipulate shared state. Another approach could be to update shared state hoping it would complete without contention/interference from other participants. In case contention is detected, the update operation can be failed, and if desired, reattempted later. We'll see how this approach is supported by hardware later.

Atomic vs **volatile**#

Short of locking, we have volatile variables that promise the same visibility guarantees as a lock, however, **volatile variables can't be used for:**

- **Executing compound actions**, e.g. Decrementing a counter involves fetching the counter value, decrementing it and then writing the updated value for a total of three steps.
- When the value of a variable depends on another or the new value of a variable depends on its older value.

The above limitations are addressed by atomic classes, which offer similar memory visibility guarantees as volatile variables and also allow operations such as read-modify-write to be executed atomically. As an aside, consider the following snippet:

```
volatile AtomicInteger atomicInteger = new AtomicInteger();
```

Note that marking the `AtomicInteger` variable above `volatile` isn't superfluous and implies that when the variable `atomicInteger` is updated to a new reference, the updated value `atomicInteger` holds will be observed by all threads that read the variable. In the absence of `volatile` the `atomicInteger` variable's value (which points to a memory location) may get cached by a processor and the new object the variable points to after the update, may not be visible to the processor that cached the old value.

Atomic variables can also be thought of as "better volatiles". Atomic variables make ideal counters, sequence generators, and variables that accumulate running statistics. They offer the same memory semantics as volatile variables with additional support for atomic updates and may be better choices than volatile variables in most circumstances.

Atomic Processor Instructions#

Modern processors have instructions that can atomically execute compound operations offering a compromise between locking and volatile variables. Hardware support for concurrency is ubiquitous in present-day processors and the most well-known of these instructions is the *Compare and Swap* instruction or CAS for short. The CAS instruction is the secret sauce behind atomically executing compound operations.

Compare and Swap#

In general, the CAS instruction has three operands:

1. A memory location, say M, representing the variable we desire to manipulate.
2. An expected value for the variable, say A. This is the latest value seen for the variable.
3. The new value, say B, which we want the variable to update to.

CAS instruction works by performing the following actions atomically:

1. Check the latest value of the memory location M.
2. If the memory location has a value other than A, then it implies that another thread changed the variable since the last time we examined it and therefore the requested update operation should be aborted.
3. If the variable's value is indeed A, then it implies that no other thread has had a chance to change the variable to a different value than A, since we last examined the variable's value and therefore we can proceed to update the variable/memory location to the new value B.

CAS takes the *optimistic approach* when performing an update on a shared variable. Expressed in prose, the instruction says *I have seen and expect the variable's value to be A, if that is still true, update the variable to the new value B, otherwise fail my request and let me know. When multiple threads invoke CAS to manipulate a shared variable, only a single thread succeeds and the rest fail.* However, the crucial difference between CAS and locking is that with CAS the threads that fail executing the CAS command have a choice to retry or go do some other useful work, unlike locking where all the threads unsuccessful in acquiring the lock will block. This may sound trivial but can improve throughput and performance many fold.

The idiomatic usage of CAS usually takes the form of reading the value *A* of a shared variable, deriving a new value *B* from the value *A*, and finally

invoking CAS to update the variable from *A* to *B* if it hasn't been changed to another value in the meantime.

ABA Problem#

The astute reader would recognize that CAS succeeds even if the value of a shared variable is changed from *A* to *B* and then back to *A*. Consider the following sequence:

1. A thread T1 reads the value of a shared variable as *A* and desires to change it to *B*. After reading the variable's value, thread
2. T1 undergoes a context switch. Another thread, T2 comes along, changes the value of the shared variable from *A* to *B* and then back to *A* from *B*.
3. Thread T1 is scheduled again for execution and invokes CAS with *A* as the expected value and *B* as the new value. CAS succeeds since the current value of the variable is *A*, even though it changed to *B* and then back to *A* in the time thread T1 was context switched.

For some algorithms the *ABA* problem may not be an issue but for others changing the value from *A* to *B* and then back to *A* may require re-executing some step(s) of an algorithm. This problem usually occurs when a program manages its own memory rather than leaving it to the Garbage Collector. For example, you may want to recycle the nodes in your linked list for performance reasons. Thus noting that the head of the list still references the same node, may not necessarily imply that the list wasn't changed. One solution to this problem is to attach a version number with the value, i.e. instead of storing the value as *A*, we store it as a pair (*A*, *V1*). Another thread can change the value to (*B*, *V1*) but when it changes it back to *A* the associated version is different i.e. (*A*, *V2*). In this way, a collision can be detected. There are two classes in Java that can be used to address the ABA problem:

1. [AtomicStampedReference](#) (please see this class in our reference section for detailed explanation of ABA problem)
2. [AtomicMarkableReference](#)

More on Atomics

Learn about the different categories of atomic classes and their performance in comparison to locks.

We'll cover the following

- Taxonomy of atomic classes
- Atomics are not primitives
- Performance of atomics vs locks
- Keeping state thread local

Taxonomy of atomic classes#

There are a total of sixteen atomic classes divided into four groups:

1. Scalars
2. Field updaters
3. Arrays
4. Compound variables

Most well-known and commonly used are the scalar ones such as `AtomicInteger`, `AtomicLong`, `AtomicReference`, which support the CAS (compare-and-set). Other primitive types such as double and float can be simulated by casting `short` or `byte` values to and from `int` and using methods `floatToIntBits()` and `doubleToLongBits()` for floating point numbers. Atomic scalar classes extend from `Number` and don't redefine `hashCode()` or `equals()`.

Atomics are not primitives#

The following snippet highlights these differences between `Integer` and `AtomicInteger`. Note, that the `Integer` class has the same hashcode for the same integer value but that's not the case for `AtomicInteger`. Thus `Atomic*` scalar classes are unsuitable as keys for collections that rely on hashcode.

```
import java.util.concurrent.atomic.AtomicInteger;

class Demonstration {
    public static void main( String args[] ) {
        AtomicInteger atomicFive = new AtomicInteger(5);
        AtomicInteger atomicAlsoFive = new AtomicInteger(5);

        System.out.println("atomicFive.equals(atomicAlsoFive) : " +
atomicFive.equals(atomicAlsoFive));
        System.out.println("atomicFive.hashCode() == atomicAlsoFive.hashCode() : " +
(atomicFive.hashCode() == atomicAlsoFive.hashCode()));
    }
}
```

```

Integer integer1 = new Integer(23235);
Integer integer2 = new Integer(23235);

System.out.println("integer1.equals(integer2) : " + integer1.equals(integer2));
System.out.println("integer1.hashCode() == integer2.hashCode() : " + (integer1.hashCode()
== integer2.hashCode()));

}

}

```

Atomics provides the users with an option to back off when faced with contention. For instance the `AtomicInteger` class has a `compareAndSet()` method that returns false if the operation doesn't succeed. The invoking thread now has the opportunity to either retry the operation immediately or use a custom retry strategy. In essence, responding to contention or *contention management* is pushed to the invoking thread and the JVM doesn't make a decision for the caller, as it does in case of locking by suspending the thread.

Performance of atomics vs locks#

In the case of a single thread, i.e. zero contention environment, an operation that relies on CAS (e.g. updating an `AtomicInteger`) will be faster than an operation that involves locking first. On single CPU machines, CAS operations almost always succeed other than in the very rare case of a thread being interrupted in the middle of a read-modify-write operation. Even with moderate contention, CAS operations are faster as they avoid thread suspension and resumption, which locking solutions must deal with.

In benchmark tests, it has been observed that atomics perform better than locks under low to moderate contention, which is representative of real-life programs. However, in a highly contended environment, the majority of threads will waste CPU cycles retrying CAS operations but using locks in the same situation would have the threads suspended and then resumed later. Granted, thread suspension/resumption incurs overhead but at some point the CAS-retry expense dwarfs the overhead of suspending and resuming threads. In reality such high contention is unlikely and atomics are always preferable over lock-based solutions.

We can conduct a crude test to measure the performance of an `AtomicInteger` counter versus an ordinary counter. The test involves creating a thread pool of ten threads and then having each thread increment the same counter a million times so that the counter value reaches ten million at the end of the test. We time the two scenarios in the widget below:

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

```

import java.util.concurrent.atomic.*;

class Demonstration {

    static AtomicInteger counter = new AtomicInteger(0);
    static int simpleCounter = 0;

    public static void main( String args[] ) throws Exception {
        test(true);
        test(false);
    }

    synchronized static void incrementSimpleCounter() {
        simpleCounter++;
    }

    static void test(boolean isAtomic) throws Exception {
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        long start = System.currentTimeMillis();

        try {
            for (int i = 0; i < 10; i++) {
                executorService.submit(new Runnable() {
                    @Override
                    public void run() {
                        for (int i = 0; i < 1000000; i++) {

                            if (isAtomic) {
                                counter.incrementAndGet();
                            } else {
                                incrementSimpleCounter();
                            }
                        }
                    });
            }
        } finally {
            executorService.shutdown();
            executorService.awaitTermination(1, TimeUnit.HOURS);
        }

        long timeTaken = System.currentTimeMillis() - start;
        System.out.println("Time taken by " + (isAtomic ? "atomic integer counter " : "integer
counter ") + timeTaken + " milliseconds.");
    }
}

```

Keeping state thread local#

Finally, the best choice for scalability and performance is to share as little state as possible among threads. Keeping variables and state, thread local results in maximum performance and elimination of contention. Even though atomics achieve better scalability than using locks, choosing not to share any state among threads will result in the best scalability.

Non-blocking Synchronization

Learn about non-blocking synchronization and thread-safe data structures that can be built using them.

We'll cover the following

- [Introduction](#)
- [Nonblocking counter](#)
 - [CAS-based counter performance](#)
- [Conclusion](#)

Introduction#

Much of the performance improvements seen in classes such as [Semaphore](#) and [ConcurrentLinkedQueue](#) versus their synchronized equivalents come from the use of atomic variables and non-blocking synchronization. Non-blocking algorithms use machine-level atomic instructions such as compare-and-swap instead of locks to provide data integrity when multiple threads access shared resources. Non-blocking algorithms are harder to design and implement but out perform lock-based alternatives in terms of liveness and scalability. As the name suggests, non-blocking algorithms don't block when multiple threads contend for the same data, and as a consequence greatly reduce scheduling overhead. These algorithms don't suffer from deadlocks, liveness issues and individual thread failures. More formally:

- An algorithm is called non-blocking if the failure or suspension of a thread doesn't cause the failure or suspension of another thread.
- An algorithm is called lock free if at every step of the algorithm some thread participant of the algorithm can make progress.

Nonblocking counter#

Designing a thread-safe counter would require using locks so that threads don't step over each other. An increment operation on a `long` variable consists of three steps. Fetching the current value, incrementing it and then writing it back. All three have to be executed atomically to achieve thread-safety. A lock-based implementation of the lock appears below:

```
class LockBasedCounter {  
    private long value;  
  
    public synchronized long getValue() {  
        return value;  
    }  
  
    // When multiple threads attempt to invoke the method at the same time,  
    // only one is allowed to do so while the rest are suspended  
    public synchronized void increment() {  
        value++;  
    }  
}
```

Locking comes with its baggage and we can design a counter that doesn't cause threads to be suspended in the face of contention. To build such a counter, we'll need the hardware to support the CAS instruction. For our purposes we'll write a class `SimulatedCAS` that imitates the CAS instruction. The listing for this class along with comments appears in the widget below:

```
public class SimulatedCAS {  
  
    // Let's assume for simplicity our value is a long  
    private long value = 0;  
  
    // constructor to initialize the value  
    public SimulatedCAS(long initialValue) {  
        value = initialValue;  
    }  
  
    synchronized long getValue() {  
        return value;  
    }  
  
    // The synchronized keyword causes all the steps in this method to execute  
    // atomically, which is akin to simulating the compare and swap processor  
    // instruction. The behavior of the function is as follows:  
    //  
    // 1. Return the expectedValue if the CAS instruction completes successfully, i.e.  
    //     the newValue is written.  
    // 2. Return the current value if the CAS instruction doesn't complete successfully  
    //  
    // The method is setup such that when expectedValue equals the return value  
    // the caller can assume success.
```

```

synchronized long compareAndSwap(long expectedValue, long newValue) {

    if (value == expectedValue) {
        value = newValue;
        return expectedValue;
    }

    // return whatever is the current value
    return value;
}

// This method uses the compareAndSwap() method to indicate if the CAS
// instruction completed successfully or not.
synchronized boolean compareAndSet(long expectedValue, long newValue) {
    return compareAndSwap(expectedValue, newValue) == expectedValue;
}
}

```

The compare and swap (CAS) functionality is also implemented by some processors as a pair of instructions *loadlinked and store-conditional*. Using the [SimulatedCAS](#) class we can now construct a non-blocking counter that doesn't block threads when multiple of them attempt to manipulate shared state. The listing for the class [NonblockingCounter](#) has comments describing the operation of the counter

```

public class NonblockingCounter {

    private SimulatedCAS count = new SimulatedCAS(0);

    public long get() {
        return count.getValue();
    }

    public void increment() {

        long currentCount;

        // The loop exits when the return value from CAS is equal to our
        // expectedValue. Otherwise we keep looping until we are successful
        do {
            currentCount = count.getValue();
        } while (currentCount != count.compareAndSwap(currentCount, currentCount + 1));

    }
}

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

        NonblockingCounter counter = new NonblockingCounter();

        counter.increment();
    }
}

```

```

        counter.increment();
        counter.increment();

        System.out.println(counter.get());
    }
}

public class SimulatedCAS {

    // Let's assume for simplicity our value is a long
    private long value = 0;

    // constructor to initialize the value
    public SimulatedCAS(long initialValue) {
        value = initialValue;
    }

    synchronized long getValue() {
        return value;
    }

    // The synchronized keyword causes all the steps in this method to execute
    // atomically, which is akin to simulating the compare and swap processor
    // instruction. The behavior of the function is as follows:
    //
    // 1. Return the expectedValue if the CAS instruction completes successfully, i.e.
    //    the newValue is written.
    // 2. Return the current value if the CAS instruction doesn't complete successfully
    //
    // The method is setup such that when expectedValue equals the return value
    // the caller can assume success.
    synchronized long compareAndSwap(long expectedValue, long newValue) {

        if (value == expectedValue) {
            value = newValue;
            return expectedValue;
        }

        // return whatever is the current value
        return value;
    }

    // This method uses the compareAndSwap() method to indicate if the CAS
    // instruction completed successfully or not.
    synchronized boolean compareAndSet(long expectedValue, long newValue) {
        return compareAndSwap(expectedValue, newValue) == expectedValue;
    }
}

```

CAS-based counter performance#

The performance of CAS instruction varies across processors and architectures and though it may seem that a CAS-based counter may perform poorly in comparison to a lock-based counter, the reality is otherwise. In practice CAS-based locks outperform lock-based counters when there is no contention (as the thread doesn't go through the process of acquiring a lock) and often when there is low to moderate contention. Acquiring a lock usually involves at least one CAS operation and peripheral lock-related housekeeping tasks, which implies more work is done by a lock-based counter in the best case of zero contention compared to a CAS-based counter.

Conclusion#

There are well-known non-blocking algorithms for commonly used data structures such as hashtables, priority queues, stacks, linked-lists etc. However, designing non-blocking algorithms is much more complex than lock-based alternatives. Generally, non-blocking algorithms skirt many of the vices associated with lock-based approaches such as deadlocks or priority inversion, however, threads participating in a non-blocking algorithm can still experience starvation or livelocks as they may perform repeated retries without success.

Miscellaneous Topics

We'll cover the following

- o [Lock Fairness](#)
- o [Thread Pools](#)

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.

Java Memory Model

This lesson lays out the ground work for understanding the Java Memory Model.

We'll cover the following

- o [Within-Thread as-if-serial](#)

A memory model is defined as the set of rules according to which the compiler, the processor or the runtime is permitted to reorder memory operations. Reordering operations allow compilers and the like to apply optimizations that can result in better performance. However, this freedom can wreak havoc in a multithreaded program when the memory model is not well-understood with unexpected program outcomes.

Consider the below code snippet executed in our **main** thread. Assume the application also spawns a couple of other threads, that'll execute the method `runMethodForOtherThreads()`

```
1. public class BadExample {
2.
3.     int myVariable = 0;
4.     boolean neverQuit = true;
5.
6.     public void runMethodForMainThread() {
7.
8.         // Change the variable value to lucky 7
9.         myVariable = 7;
10.    }
11.
12.    public void runMethodForOtherThreads() {
13.
14.        while (neverQuit) {
15.            System.out.println("myVariable : " + myVariable);
16.        }
17.    }
18. }
```

Now you would expect that the other threads would see the `myVariable` value change to `7` as soon as the **main** thread executes the assignment on **line 9**. This assumption is false in modern architectures and other threads may see the change in the value of the variable `myVariable` with a delay or not at all. Below are some of the reasons that can cause this to happen

- Use of sophisticated multi-level memory caches or processor caches
- Reordering of statements by the compiler which may differ from the source code ordering
- Other optimizations that the hardware, runtime or the compiler may apply.

One likely scenario can be that the variable is updated with the new value in the processor's cache but not in the main memory. When another thread running on another core requests the variable `myVariable`'s value from the memory, it still sees the stale value of `0`. This is a specific example of the **cache coherence** problem. Different processor architectures have different policies as to when an individual processor's cache is reconciled with the main memory.

The above discussion then begets the question: "*Under what circumstances does a thread reading the `myVariable` value would see the updated value of `7`?*" This can be answered by understanding the Java memory model (JMM).

Within-Thread as-if-serial#

The Java language specification (JLS) mandates the JVM to maintain **within-thread as-if-serial** semantics. What this means is that, as long as the result of the program is exactly the same if it were to be executed in a strictly sequential environment (think single thread, single processor) then the JVM is free to undertake any optimizations it may deem necessary. Over the years, much of the performance improvements have come from these clever optimizations as clock rates for processors become harder to increase. However, when data is shared between threads, these very optimizations can result in concurrency errors and the developer needs to inform the JVM through synchronization constructs of when data is being shared.

Let's see in the next lesson how these optimizations can give surprising results in multithreaded scenarios.

Reordering Effects

This lesson discusses the compiler, runtime or hardware optimizations that can cause reordering of program instructions

We'll cover the following

- Affect of memory architectures

Take a look at the following program and try to come up with all the possible outcomes for the variables `ping` and `pong`.

```
class Demonstration {  
    public static void main( String args[] ) throws Exception {  
        (new ReorderingExample()).reorderTest();  
    }  
}  
  
class ReorderingExample {  
  
    private int ping = 0;  
    private int pong = 0;  
    private int foo = 0;  
    private int bar = 0;  
  
    public void reorderTest() throws InterruptedException {  
  
        Thread t1 = new Thread(new Runnable() {  
  
            public void run() {  
                foo = 1;  
                ping = bar;  
            }  
        });  
  
        Thread t2 = new Thread(new Runnable() {  
  
            public void run() {  
                bar = 1;  
                pong = foo;  
            }  
        });  
  
        t1.start();  
        t2.start();  
        t1.join();  
        t2.join();  
        System.out.println(ping + " " + pong);  
  
    }  
}
```

Most folks would come up with the following possible outcomes:

- 1 and 1
- 1 and 0
- 0 and 1

However, it might surprise many but the program can very well print **0** **and 0!** How is that even possible? Think from the point of view of a compiler, it sees the following instructions for **thread t1's run()** method:

```
bar = 1;  
pong = foo;
```

The compiler doesn't know that the variable **bar** is being used by another thread so it may take the liberty to **reorder** the statements like so:

```
pong = foo;  
bar = 1;
```

The two statements don't have a dependence on each other in the sense that they are working off of completely different variables. For performance reasons, the compiler may decide to switch their ordering. Other forces are also at play, for instance, the value of one of the variables may get flushed out to the main memory from the processor cache but not for the other variable.

Note that with the reordering of the statements the JVM still is able to honor the *within-thread as-if-serial* semantics and is completely justified to move the statements around. Such performance and optimization tricks by the compiler, runtime or hardware catch unsuspecting developers off-guard and lead to bugs which are very hard to reproduce in production.

Affect of memory architectures#

Java is touts the famous **code once, run anywhere** mantra as one of its strengths. However, this isn't possible without Java shielding us from the vagrancies of the multitude of memory architectures that exist in the wild. For instance, the frequency of reconciling a processor's cache with the main memory depends on the processor architecture. A processor may relax its memory coherence guarantees in favor of better performance. The architecture's memory model specifies the guarantees a program can expect from the memory model. It will also specify instructions required to get additional memory coordination guarantees when data is being shared among threads. These instructions are usually called *memory fences* or

barriers but the Java developer can rely on the JVM to interface with the underlying platform's memory model through its own memory model called JMM (Java Memory Model) and insert these platform memory specific instructions appropriately. Conversely, the JVM relies on the developer to identify when data is shared through the use of proper synchronization.

The happens-before Relationship

This lesson continues the in-depth discussion of Java memory model

We'll cover the following

- o [Total Order](#)
- o [Partial Order](#)
- o [Java Memory Model Details](#)

The JMM defines a ***partial ordering on all actions within a program***. This might sound like a loaded statement so bear with me as I explain below.

Total Order#

You are already familiar with total ordering, the sequence of natural numbers i.e. 1, 2, 3 4, is a total ordering. Each element is either greater or smaller than any other element in the set of natural numbers (**Totality**). If **2 < 4** and **4 < 7** then we know that **2 < 7** necessarily (**Transitivity**). And finally if **3 < 5** then 5 can't be less than 3 (**Asymmetry**).

Partial Order#

Elements of a set will exhibit *partial ordering* when they possess transitivity and asymmetry but not totality. As an example think about your family tree. Your father is your ancestor, your grandfather is your father's ancestor. By transitivity, your grandfather is also your ancestor. However, your father or grandfather aren't ancestors of your mother and in a sense they are incomparable.

Java Memory Model Details#

The compiler in the spirit of optimization is free to reorder statements however it must make sure that the outcome of the program is the same as without reordering. The sources of reordering can be numerous. Some examples include:

- If two fields **X** and **Y** are being assigned but don't depend on each other, then the compiler is free to reorder them
- Processors may execute instructions out of order under some circumstances
- Data may be juggled around in the registers, processor cache or the main memory in an order not specified by the program e.g. **Y** can be flushed to main memory before **X**.

Note that all these reorderings may happen behind the scenes in a single-threaded program but the program sees no ill-effects of these reorderings as the JMM guarantees that the outcome of the program would be the same as if these reorderings never happened.

However, when multiple threads are involved then these reorderings take on an altogether different meaning. Without proper synchronization, these same optimizations can wreak havoc and program output would be unpredictable.

The JMM is defined in terms of *actions* which can be any of the following

- read and writes of variables
- locks and unlocks of monitors
- starting and joining of threads

The JMM enforces a ***happens-before*** ordering on these actions. When an action A *happens-before* an action B, it implies that A is guaranteed to be ordered before B and visible to B. Consider the below program

```
public class ReorderingExample {  
    int x = 3;  
    int y = 7;  
    int a = 4;  
    int b = 9;  
    Object lock1 = new Object();  
    Object lock2 = new Object();  
  
    public void writerThread() {  
        // BLOCK#1  
        // The statements in block#1 and block#2 aren't dependent  
        // on eachother and the two blocks can be reordered by the  
        // compiler  
        x = a;
```

```

// BLOCK#2
// These two writes within block#2 can't be reordered, as
// they are dependent on eachother. Though this block can
// be ordered before block#1
y += y;
y *= y;

// BLOCK#3
// Because this block uses x and y, it can't be placed before
// the assignments to the two variables, i.e. block#1 and block#2
synchronized (lock1) {
    x *= x;
    y *= y;
}

// BLOCK#4
// Since this block is also not dependent on block#3, it can be
// placed before block#3 or block#2. But it can't be placed before
// block#1, as that would assign a different value to x
synchronized (lock2) {
    a *= a;
    b *= b;
}
}
}

```

Now note that even though all this reordering magic can happen in the background but the notion of *program order* is still maintained i.e. the final outcome is exactly the same as without the ordering.

Furthermore, **block#1** will appear to *happen-before* **block#2** even if **block#2** gets executed before. Also note that **block#2** and **block#4** have no ordering dependency on each other.

One can see that there's no partial ordering between **block#1** and **block#2** but there's a partial ordering between **block#1** and **block#3** where **block#3** must come after **block#1**.

The reordering tricks are harmless in case of a single threaded program but all hell will break loose when we introduce another thread that shares the data that is being read or written to in the `writerThread` method. Consider the addition of the following method to the previous class.

```

public void readerThread() {

    a *= 10;

    // BLOCK#4
    // Moved out to here from writerThread method
    synchronized (lock2) {
        a *= a;
        b *= b;
    }
}

```

```
    }  
}
```

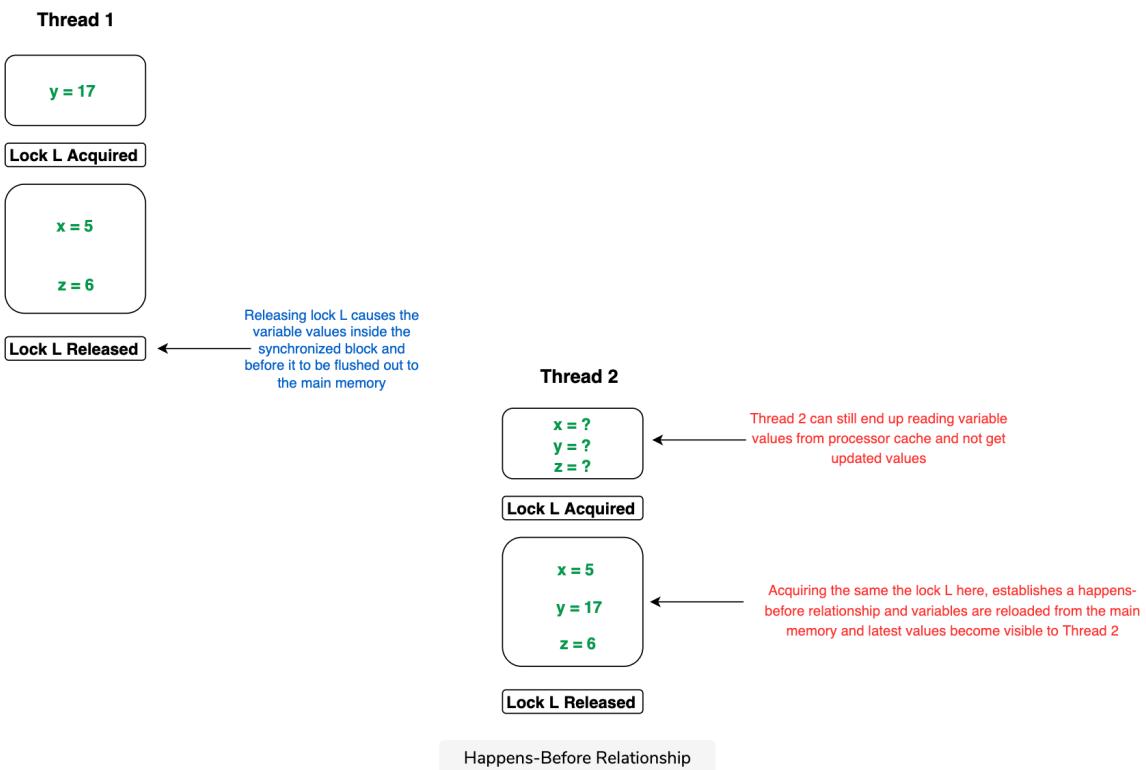
Note we moved out **block#4** into the new method `readerThread`. Say if the `readerThread` runs to completion, it is possible for the `writerThread` to never see the updated value of the variable `a` as it may never have been flushed out to the main memory, where the `writerThread` would attempt to read from. There's no *happens before* relationship between the two code snippets executed in two different threads!

To make sure that the changes done by one thread to shared data are visible immediately to the next thread accessing those same variables, we must establish a *happens-before* relationship between the execution of the two threads. A happens before relationship can be established in the following ways.

- Each action in a thread *happens-before* every action in that thread that comes later in the program's order. However, for a single-threaded program, instructions can be reordered but the semantics of the program order is still preserved.
- An unlock on a monitor *happens-before* every subsequent lock on that same monitor. The `synchronization` block is equivalent of a monitor.
- A write to a volatile field *happens-before* every subsequent read of that same volatile.
- A call to `start()` on a thread *happens-before* any actions in the started thread.
- All actions in a thread *happen-before* any other thread successfully returns from a `join()` on that thread.
- The constructor for an object *happens-before* the start of the finalizer for that object
- A thread interrupting another thread *happens-before* the interrupted thread detects it has been interrupted.

This implies that any memory operations which were visible to a thread before exiting a synchronized block are visible to any thread after it enters a synchronized block protected by the same monitor, since all the memory operations happen before the release, and the release happens before the acquire. Exiting a synchronized block causes the cache to be flushed to the main memory so that the writes made by the exiting thread are visible to other threads. Similarly, entering a synchronized block has the effect of invalidating the local processor cache and reloading of variables from the main memory so that the entering thread is able to see the latest values.

In our `readerThread` if we synchronize on the same lock object as the one we synchronize on in the `writerThread` then we would establish a *happens-before* relationship between the two threads. Don't confuse it to mean that one thread executes before the other. All it means is that when `readerThread` releases the monitor, up till that point, whatever shared variables it has manipulated will have their latest values visible to the `writerThread` as soon as it acquires the **same** monitor. If it acquires a different monitor then there's no happens-before relationship and it may or may not see the latest values for the shared variables



Interview Practice:

Classical synchronization problem involving a limited size buffer which can have items added to it or removed from it by different producer and consumer threads. This problem is known by different names: consumer producer problem, bounded buffer problem or blocking queue problem.

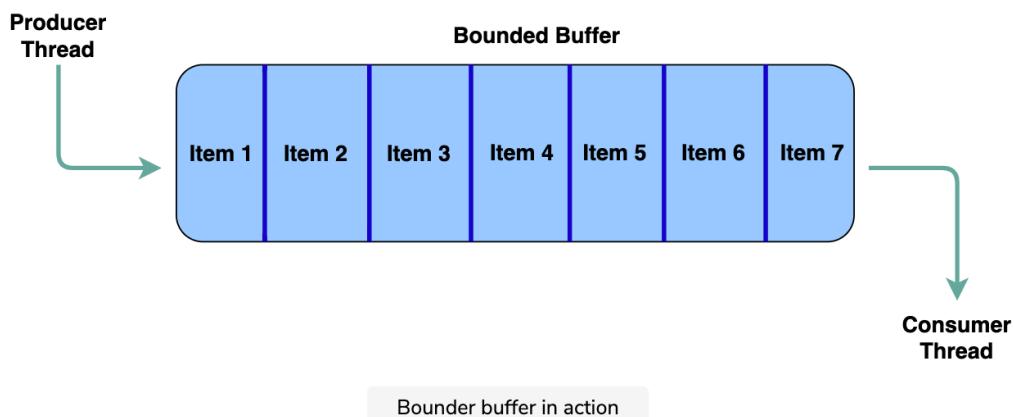
We'll cover the following

- o [Problem Statement](#)
- o [Solution](#)
- o [Complete Code](#)

- o [Follow-up Question](#)

Problem Statement#

A blocking queue is defined as a queue which blocks the caller of the enqueue method if there's no more capacity to add the new item being enqueued. Similarly, the queue blocks the dequeue caller if there are no items in the queue. Also, the queue notifies a blocked enqueueing thread when space becomes available and a blocked dequeuing thread when an item becomes available in the queue.



Solution#

Our queue will have a finite size that is passed in via the constructor. Additionally, we'll use an array as the data structure for backing our queue. Furthermore, we'll expose the APIs `enqueue` and `dequeue` for our blocking queue class. We'll also need a **head** and a **tail** pointer to keep track of the front and back of the queue and a size variable to keep track of the queue size at any given point in time. Given this, the skeleton of our blocking queue class would look something like below:

```
public class BlockingQueue<T> {

    T[] array;
    int size = 0;
    int capacity;
    int head = 0;
    int tail = 0;

    public BlockingQueue(int capacity) {
        array = (T[]) new Object[capacity];
        this.capacity = capacity;
    }

    public void enqueue(T item) {
```

```

    }

    public T dequeue() {
}
}

```

Let's start with the **enqueue** method. If the current `size of the queue == capacity` then we know we'll need to block the caller of the method. We can do so by appropriately calling **wait()** method in a while loop. The while loop is conditioned on the size of the queue being equal to the max capacity. The loop's predicate would become false, as soon as, another thread performs a dequeue.

Note that whenever we test for the value of the **size** variable, we also need to make sure that no other thread is manipulating the size variable. This can be achieved by the synchronized keyword as it'll only allow a single thread to invoke the enqueue/dequeue methods on the queue object.

Finally, as the queue grows, it'll reach the end of our backing array, so we need to reset the tail of the queue back to zero. Notice that since we only proceed to enqueue an item when `size of queue < capacity` we are guaranteed that tail would not be overwriting an existing item.

```

public synchronized void enqueue(T item) throws InterruptedException {

    // wait for queue to have space
    while (size == capacity) {
        wait();
    }

    // reset tail to the beginning if the tail is already
    // at the end of the backing array
    if (tail == capacity) {
        tail = 0;
    }

    // place the item in the array
    array[tail] = item;
    size++;
    tail++;

    // don't forget to notify any other threads waiting on
    // a change in value of size. There might be consumers
    // waiting for the queue to have atleast one element
    notifyAll();
}

```

Note that in the end we are calling `notifyAll()` method. Since we just added an item to the queue, it is possible that a consumer thread is blocked in the

dequeue method of the queue class waiting for an item to become available so it's necessary we send a signal to wake up any waiting threads.

If no thread is waiting, then the signal will simply go unnoticed and be ignored, which wouldn't affect the correct working of our class. This would be an instance of **missed signal** that we have talked about earlier.

Now let's design the `dequeue` method. Similar to the enqueue method, we need to block the caller of the dequeue method if there's nothing to dequeue i.e. `size == 0`

We need to reset head of the queue back to zero in-case it's pointing past the end of the array. We need to decrement the size variable too since the queue will now have one less item.

Finally, we remember to call `notifyAll()` since if the queue were full then there might be producer threads blocked in the enqueue method. This logic in code appears as below:

```
public synchronized T dequeue() throws InterruptedException {  
    T item = null;  
  
    // wait for atleast one item to be enqueued  
    while (size == 0) {  
        wait();  
    }  
  
    // reset head to start of array if its past the array  
    if (head == capacity) {  
        head = 0;  
    }  
  
    // store the reference to the object being dequeued  
    // and overwrite with null  
    item = array[head];  
    array[head] = null;  
    head++;  
    size--;  
  
    // don't forget to call notify, there might be another thread  
    // blocked in the enqueue method.  
    notifyAll();  
  
    return item;  
}
```

We see the dequeue method is analogous to enqueue method. Note that we could have eliminated lines 17 & 18 and instead just returned the following:

```
return array[head-1];
```

but for better readability we choose to expand this operation into two lines.

Complete Code#

The full code for the blocking queue appears below.

```
class Demonstration {  
    public static void main( String args[] ) throws Exception{  
        final BlockingQueue<Integer> q = new BlockingQueue<Integer>(5);  
  
        Thread t1 = new Thread(new Runnable() {  
  
            @Override  
            public void run() {  
                try {  
                    for (int i = 0; i < 50; i++) {  
                        q.enqueue(new Integer(i));  
                        System.out.println("enqueued " + i);  
                    }  
                } catch (InterruptedException ie) {  
  
                }  
            }  
        });  
  
        Thread t2 = new Thread(new Runnable() {  
  
            @Override  
            public void run() {  
                try {  
                    for (int i = 0; i < 25; i++) {  
                        System.out.println("Thread 2 dequeued: " + q.dequeue());  
                    }  
                } catch (InterruptedException ie) {  
  
                }  
            }  
        });  
  
        Thread t3 = new Thread(new Runnable() {  
  
            @Override  
            public void run() {  
                try {  
                    for (int i = 0; i < 25; i++) {  
                        System.out.println("Thread 3 dequeued: " + q.dequeue());  
                    }  
                }  
            }  
        });  
    }  
}
```

```

        } catch (InterruptedException ie) {
            }
        }
    });

t1.start();
Thread.sleep(4000);
t2.start();

t2.join();

t3.start();
t1.join();
t3.join();
}
}

// The blocking queue class
class BlockingQueue<T> {

    T[] array;
    Object lock = new Object();
    int size = 0;
    int capacity;
    int head = 0;
    int tail = 0;

    @SuppressWarnings("unchecked")
    public BlockingQueue(int capacity) {
        // The casting results in a warning
        array = (T[]) new Object[capacity];
        this.capacity = capacity;
    }

    public void enqueue(T item) throws InterruptedException {

        synchronized (lock) {

            while (size == capacity) {
                lock.wait();
            }

            if (tail == capacity) {
                tail = 0;
            }

            array[tail] = item;
        }
    }
}

```

```

        size++;
        tail++;
        lock.notifyAll();
    }
}

public T dequeue() throws InterruptedException {

    T item = null;
    synchronized (lock) {

        while (size == 0) {
            lock.wait();
        }

        if (head == capacity) {
            head = 0;
        }

        item = array[head];
        array[head] = null;
        head++;
        size--;

        lock.notifyAll();
    }

    return item;
}
}

```

The test case in our example creates two dequeuer threads and one enqueueer thread. The enqueue-er thread initially fills up the queue and gets blocked, till the dequeuer threads start off and remove elements from the queue. The output would show enqueueing and dequeuing activity interleaved after the first 5 enqueues.

Follow-up Question#

In both the `enqueue()` and `dequeue()` methods we use the `notifyAll()` method instead of the `notify()` method. The reason behind the choice is very crucial to understand. Consider a situation with two producer threads and one consumer thread all working with a queue of size one. It's possible that when an item is added to the queue by one of the producer threads, the other two threads are blocked waiting on the condition variable. If the producer thread after adding an item invokes `notify()` it is possible that the other producer thread is chosen by the system to resume execution. The

woken-up producer thread would find the queue full and go back to waiting on the condition variable, causing a deadlock. Invoking `notifyAll()` assures that the consumer thread also gets a chance to wake up and resume execution.

... continued

This lesson explains how to solve the producer-consumer problem using a mutex.

We'll cover the following

- o Busy wait solution using Lock
- o Faulty Implementation
 - Incorrect `dequeue()` implementation
 - Incorrect `enqueue()` implementation

Busy wait solution using `Lock`#

In the previous lesson, we solved the consumer producer problem using the `synchronized` keyword, which is equivalent of a monitor in Java. Let's see how the implementation would look like, if we were restricted to using a mutex. There's no direct equivalent of a theoretical mutex in Java as each object has an implicit monitor associated with it. For this question, we'll use an object of the `Lock` class and pretend it doesn't expose the `wait()` and `notify()` methods and only provides mutual exclusion similar to a theoretical mutex. Without the ability to wait or signal the implication is, a blocked thread will constantly poll in a loop for a predicate/condition to become true before making progress. This is an example of a busy-wait solution.

Let's start with the `enqueue()` method. If the current `size of the queue == capacity` then we know we need to block the caller of the method until the queue has space for a new item. Since a mutex only allows locking, we give up the mutex at this point. The logic is shown below.

```
lock.lock();
while (size == capacity) {
    // Release the mutex to give other threads
    lock.unlock();
    // Reacquire the mutex before checking the
    // condition
    lock.lock();
}

if (tail == capacity) {
```

```

        tail = 0;
    }

    array[tail] = item;
    size++;
    tail++;
    lock.unlock();
}

```

The most important point to realize in the above code is the weird-looking while loop construct, where we release the lock and then immediately attempt to reacquire it. Convince yourself that whenever we test the while loop condition `size == capacity`, we do so while holding the mutex! Also, it may not be immediately obvious but a different thread can acquire the mutex just when a thread releases the mutex and attempts to reacquire it within the while loop. Lastly, we modify the `array` variable only when holding the mutex.

We also need to manage the `tail` as the queue grows. Once it reaches the end of our backing array, we reset it to zero. Realize that since we only proceed to add an item when `size < maxSize` we are guaranteed that `tail` will never overwrite an existing item.

Now let us see the code for the `dequeue()` method which is analogous to the `enqueue()` one.

```

T item = null;

lock.lock();
while (size == 0) {
    lock.unlock();
    lock.lock();
}

if (head == capacity) {
    head = 0;
}

item = array[head];
array[head] = null;
head++;
size--;

lock.unlock();
return item;
}

```

Again note that we always test for the condition `size == 0` when holding the lock. Additionally, all shared state is manipulated in mutual exclusion. Additionally, we reset `head` of the queue back to zero in case it's pointing past the end of the array. We need to decrement the `size` variable too since the queue will now have one less item. The complete code appears in the widget below. It also runs a simulation of several producers and consumers

that constantly write and retrieve from an instance of the blocking queue, for one second.

Main.java:

```
class Demonstration {  
    public static void main( String args[] ) throws InterruptedException {  
        final BlockingQueueWithMutex<Integer> q = new BlockingQueueWithMutex<Integer>(5);  
  
        Thread producer1 = new Thread(new Runnable() {  
            public void run() {  
                try {  
                    int i = 1;  
                    while (true) {  
                        q.enqueue(i);  
                        System.out.println("Producer thread 1 enqueued " + i);  
                        i++;  
                    }  
                } catch (InterruptedException ie) {  
                }  
            }  
        });  
  
        Thread producer2 = new Thread(new Runnable() {  
            public void run() {  
                try {  
                    int i = 5000;  
                    while (true) {  
                        q.enqueue(i);  
                        System.out.println("Producer thread 2 enqueued " + i);  
                        i++;  
                    }  
                } catch (InterruptedException ie) {  
                }  
            }  
        });  
  
        Thread producer3 = new Thread(new Runnable() {  
            public void run() {  
                try {  
                    int i = 100000;  
                    while (true) {  
                        q.enqueue(i);  
                        System.out.println("Producer thread 3 enqueued " + i);  
                        i++;  
                    }  
                } catch (InterruptedException ie) {  
                }  
            }  
        });
```

```

        }
    } catch (InterruptedException ie) {

    }
}
});

Thread consumer1 = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                System.out.println("Consumer thread 1 dequeued " + q.dequeue());
            }
        } catch (InterruptedException ie) {

        }
    }
});
}

Thread consumer2 = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                System.out.println("Consumer thread 2 dequeued " + q.dequeue());
            }
        } catch (InterruptedException ie) {

        }
    }
});
}

Thread consumer3 = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                System.out.println("Consumer thread 3 dequeued " + q.dequeue());
            }
        } catch (InterruptedException ie) {

        }
    }
});
}

producer1.setDaemon(true);
producer2.setDaemon(true);
producer3.setDaemon(true);
consumer1.setDaemon(true);
consumer2.setDaemon(true);

```

```

consumer3.setDaemon(true);

producer1.start();
producer2.start();
producer3.start();

consumer1.start();
consumer2.start();
consumer3.start();

Thread.sleep(1000);
}
}

```

BlockingQueueWithMutex.java

```

BlockingQueueWithMutex.java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class BlockingQueueWithMutex<T> {
    T[] array;
    Lock lock = new ReentrantLock();
    int size = 0;
    int capacity;
    int head = 0;
    int tail = 0;

    @SuppressWarnings("unchecked")
    public BlockingQueueWithMutex(int capacity) {
        // The casting results in a warning
        array = (T[]) new Object[capacity];
        this.capacity = capacity;
    }

    public T dequeue() throws InterruptedException {
        T item = null;

        lock.lock();
        while (size == 0) {
            lock.unlock();
            lock.lock();
        }
    }
}
```

```

        if (head == capacity) {
            head = 0;
        }

        item = array[head];
        array[head] = null;
        head++;
        size--;

        lock.unlock();
        return item;
    }

    public void enqueue(T item) throws InterruptedException {

        lock.lock();
        while (size == capacity) {
            // Release the mutex to give other threads
            lock.unlock();
            // Reacquire the mutex before checking the
            // condition
            lock.lock();
        }

        if (tail == capacity) {
            tail = 0;
        }

        array[tail] = item;
        size++;
        tail++;
        lock.unlock();
    }
}

```

Faulty Implementation#

As an exercise, we reproduce the two `enqueue()` and `dequeue()` methods, without locking the mutex object when checking for the while-loop conditions. If you run the code in the widget below multiple times, some of the runs would display a dequeue value of null. We set an array index to null whenever we remove its content to indicate the index is now empty. A race condition is introduced when we check for while-loop predicate without holding a mutex.

Incorrect **dequeue()** implementation#

```
public T dequeue() {  
    T item = null;  
  
    while (size == 0) { }  
  
    lock.lock();  
    if (head == capacity) {  
        head = 0;  
    }  
  
    item = array[head];  
    array[head] = null;  
    head++;  
    size--;  
  
    lock.unlock();  
    return item;  
}
```

and,

Incorrect **enqueue()** implementation#

```
public void enqueue(T item) {  
  
    while (size == capacity) { }  
  
    lock.lock();  
    if (tail == capacity) {  
        tail = 0;  
    }  
  
    array[tail] = item;  
    size++;  
    tail++;  
    lock.unlock();  
}
```

Main.java

```
class Demonstration {  
  
    static final FaultyBlockingQueueWithMutex<Integer> q = new  
    FaultyBlockingQueueWithMutex<Integer>(5);
```

```

static void producerThread(int start, int id ) {
    while (true) {
        try {
            q.enqueue(start);
            System.out.println("Producer thread " + id + " enqueued " + start);
            start++;
            Thread.sleep(1);
        } catch (InterruptedException ie){
            // swallow exception
        }
    }
}

static void consumerThread(int id) {
    while (true) {
        try {
            System.out.println("Consumer thread " + id + " dequeued " + q.dequeue());
            Thread.sleep(1);
        } catch (InterruptedException ie){
            // swallow exception
        }
    }
}

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

    Thread producer1 = new Thread(new Runnable() {
        public void run() {
            producerThread(1, 1);
        }
    });

    Thread producer2 = new Thread(new Runnable() {
        public void run() {
            producerThread(5000, 2);
        }
    });

    Thread producer3 = new Thread(new Runnable() {
        public void run() {
            producerThread(100000, 3);
        }
    });

    Thread consumer1 = new Thread(new Runnable() {
        public void run() {
            consumerThread(1);
        }
    });
}

```

```

    });

Thread consumer2 = new Thread(new Runnable() {
    public void run() {
        consumerThread(2);
    }
});

Thread consumer3 = new Thread(new Runnable() {
    public void run() {
        consumerThread(3);
    }
});

producer1.setDaemon(true);
producer2.setDaemon(true);
producer3.setDaemon(true);
consumer1.setDaemon(true);
consumer2.setDaemon(true);
consumer3.setDaemon(true);

producer1.start();
producer2.start();
producer3.start();

consumer1.start();
consumer2.start();
consumer3.start();

Thread.sleep(20000);
}
}

```

[FaultyBlockingQueueWithMutex.java](#)

```

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class FaultyBlockingQueueWithMutex<T> {
    T[] array;
    Lock lock = new ReentrantLock();
    int size = 0;
    int capacity;
    int head = 0;
    int tail = 0;

    @SuppressWarnings("unchecked")
    public FaultyBlockingQueueWithMutex(int capacity) {
        // The casting results in a warning
        array = (T[]) new Object[capacity];
    }
}

```

```

        this.capacity = capacity;
    }

public T dequeue() {

    T item = null;

    while (size == 0) {
    }

    lock.lock();
    if (head == capacity) {
        head = 0;
    }

    item = array[head];
    array[head] = null;
    head++;
    size--;

    lock.unlock();
    return item;
}

public void enqueue(T item) {

    while (size == capacity) {
    }

    lock.lock();
    if (tail == capacity) {
        tail = 0;
    }

    array[tail] = item;
    size++;
    tail++;
    lock.unlock();
}
}

```

... continued

Solution using semaphores#

We can also implement the bounded buffer problem using a semaphore. For this problem, we'll use an instance of the [CountingSemaphore](#) that we implement in one of the later problems. A [CountingSemaphore](#) is initialized with a maximum number of permits to give out. A thread is blocked when it attempts to release the semaphore when none of the permits have been given out. Similarly, a thread blocks when attempting to acquire a semaphore that has all the permits given out. In contrast, Java's implementation of Semaphore can be signaled (released) even if none of the permits, the Java semaphore was initialized with, have been used. Java's semaphore has no upper bound and can be released as many times as desired to increase the number of permits. Before proceeding forward, it is suggested to complete the [CountingSemaphore](#) lesson.

We'll augment the [CountingSemaphore](#) class with a new constructor that takes in the maximum permits and also sets the number of permits already given out. We can use two semaphores, one [semConsumer](#) and the other [semProducer](#). The trick is to initialize [semProducer](#) semaphore with a maximum number of permits equal to the size of the buffer and set all permits as available. Each permit allows a producer thread to enqueue an item in the buffer. Since the number of permits is equal to the size of the buffer, the producer threads can only enqueue items equal to the size of the buffer and then blocks. However, the [semProducer](#) is only released/incremented by a consumer thread whenever it consumes an item. If there are no consumer threads, the producer threads will block when the buffer becomes full. In case of the consumer threads, when the buffer is empty, we would want to block any consumer threads on a [dequeue\(\)](#) call. This implies that we should initialize the [semConsumer](#) semaphore with a maximum capacity equal to the size of the buffer and set all the permits as currently given out. Let's look at the implementation of [enqueue\(\)](#) method.

```
public void enqueue(T item) throws InterruptedException {  
    semProducer.acquire();  
  
    if (tail == capacity) {  
        tail = 0;  
    }  
  
    array[tail] = item;  
    size++;  
    tail++;  
  
    semConsumer.release();  
}
```

Suppose the size of the buffer is N. If you study the code above, it should be evident that only N items can be enqueued in the [items](#) buffer. At the end of the method, we signal any consumer threads waiting on

the `semConsumer` semaphore. However, the code is not yet complete. We have only solved the problem of coordinating between the producer and the consumer threads. The astute reader would immediately realize that multiple producer threads can manipulate the code lines between the first and the last semaphore statements in the above `enqueue()` method. In our earlier implementations, we were able to guard the critical section by synchronizing on objects that ensured only a single thread is active in the critical section at a time. We need similar functionality using semaphores. Recall that we can use a binary semaphore to exercise mutual exclusion, however, any thread is free to signal the semaphore, not just the one that acquired it. We'll introduce a `semLock` semaphore that acts as a mutex. The complete version of the `enqueue()` method appears below:

```
public void enqueue(T item) throws InterruptedException {

    semProducer.acquire();
    semLock.acquire();

    if (tail == capacity) {
        tail = 0;
    }

    array[tail] = item;
    size++;
    tail++;

    semLock.release();
    semConsumer.release();
}
```

Realize that we have modeled each item in the buffer as a permit. When the buffer is full, the consumer threads have N permits to perform `dequeue()` and when the buffer is empty the producer threads have N permits to perform `enqueue()`. The code for `dequeue()` is similar and appears below:

```
public T dequeue() throws InterruptedException {

    T item = null;

    semConsumer.acquire();
    semLock.acquire();

    if (head == capacity) {
        head = 0;
    }

    item = array[head];
    array[head] = null;
    head++;
    size--;
```

```
        semLock.release();
        semProducer.release();

        return item;
    }
```

The complete code appears in the code widget below. We also include a simple test with one producer and two consumer threads.

Main.java

```
class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        final BlockingQueueWithSemaphore<Integer> q = new
BlockingQueueWithSemaphore<Integer>(5);

        Thread t1 = new Thread(new Runnable() {

            public void run() {
                try {
                    for (int i = 0; i < 20; i++) {
                        q.enqueue(new Integer(i));
                        System.out.println("enqueued " + i);
                    }
                } catch (InterruptedException ie) {

                }
            }
        });

        Thread t2 = new Thread(new Runnable() {

            public void run() {
                try {
                    for (int i = 0; i < 10; i++) {
                        System.out.println("Thread 2 dequeued: " + q.dequeue());
                    }
                } catch (InterruptedException ie) {

                }
            }
        });

        Thread t3 = new Thread(new Runnable() {

            public void run() {
                try {
                    for (int i = 0; i < 10; i++) {
                        System.out.println("Thread 3 dequeued: " + q.dequeue());
                    }
                } catch (InterruptedException ie) {

                }
            }
        });
    }
}
```

```

        }
    } catch (InterruptedException ie) {
        }
    });
}

t1.start();
Thread.sleep(4000);
t2.start();
t2.join();

t3.start();
t1.join();
t3.join();

}
}

```

[BlockingQueueWithSemaphore.java](#)

```

public class BlockingQueueWithSemaphore<T> {
    T[] array;
    int size = 0;
    int capacity;
    int head = 0;
    int tail = 0;
    CountingSemaphore semLock = new CountingSemaphore(1, 1);
    CountingSemaphore semProducer;
    CountingSemaphore semConsumer;

    @SuppressWarnings("unchecked")
    public BlockingQueueWithSemaphore(int capacity) {
        // The casting results in a warning
        array = (T[]) new Object[capacity];
        this.capacity = capacity;
        this.semProducer = new CountingSemaphore(capacity, capacity);
        this.semConsumer = new CountingSemaphore(capacity, 0);
    }

    public T dequeue() throws InterruptedException {
        T item = null;

        semConsumer.acquire();
        semLock.acquire();

```

```

        if (head == capacity) {
            head = 0;
        }

        item = array[head];
        array[head] = null;
        head++;
        size--;

        semLock.release();
        semProducer.release();

        return item;
    }

    public void enqueue(T item) throws InterruptedException {

        semProducer.acquire();
        semLock.acquire();

        if (tail == capacity) {
            tail = 0;
        }

        array[tail] = item;
        size++;
        tail++;

        semLock.release();
        semConsumer.release();
    }
}

```

[CountingSemaphore.java](#)

```

public class CountingSemaphore {

    int usedPermits = 0;
    int maxCount;

    public CountingSemaphore(int count) {
        this.maxCount = count;
    }

    public CountingSemaphore(int count, int initialPermits) {
        this.maxCount = count;
        this.usedPermits = this.maxCount - initialPermits;
    }
}

```

```
}

public synchronized void acquire() throws InterruptedException {
    while (usedPermits == maxCount)
        wait();
    notify();
    usedPermits++;
}

public synchronized void release() throws InterruptedException {
    while (usedPermits == 0)
        wait();
    usedPermits--;
    notify();
}
}
```

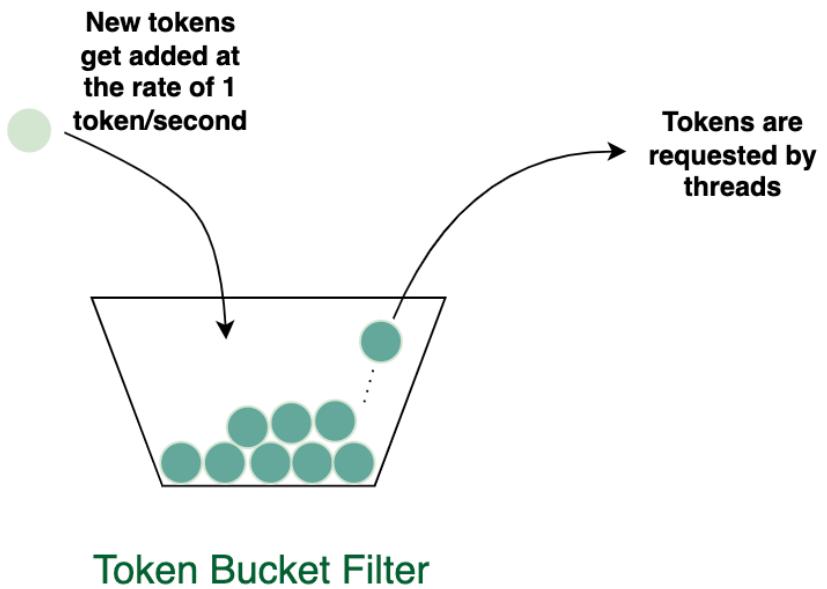
Rate Limiting Using Token Bucket Filter

Problem Statement#

This is an actual interview question asked at Uber and Oracle.

Imagine you have a bucket that gets filled with tokens at the rate of 1 token per second. The bucket can hold a maximum of N tokens. Implement a thread-safe class that lets threads get a token when one is available. If no token is available, then the token-requesting threads should block.

The class should expose an API called `getToken` that various threads can call to get a token



Solution#

This problem is a naive form of a class of algorithms called the "token bucket" algorithms. A complimentary set of algorithms is called "leaky bucket" algorithms. One application of these algorithms is shaping network traffic flows. This particular problem is interesting because the majority of candidates incorrectly start with a multithreaded approach when taking a stab at the problem. One is tempted to create a background thread to fill the bucket with tokens at regular intervals but there is a far simpler solution devoid of threads and a message to make judicious use of threads. This question tests a candidate's comprehension prowess as well as concurrency knowledge.

The key to the problem is to find a way to track the number of available tokens when a consumer requests for a token. Note the rate at which the tokens are being generated is constant. So if we know when the token bucket was instantiated and when a consumer called `getToken()` we can take the difference of the two instants and know the number of possible tokens we would have collected so far. However, we'll need to tweak our solution to account for the max number of tokens the bucket can hold. Let's start with the skeleton of our class

```
public class TokenBucketFilter {

    private int MAX_TOKENS;
    // variable to note down the latest token request.
    private long lastRequestTime = System.currentTimeMillis();
    long possibleTokens = 0;
```

```

public TokenBucketFilter(int maxTokens) {
    MAX_TOKENS = maxTokens;
}

synchronized void getToken() throws InterruptedException {
}
}

```

Note how `getToken()` doesn't return any token type ! The fact a thread can return from the `getToken` call would imply that the thread has the token, which is nothing more than a permission to undertake some action.

Note we are using `synchronized` on our `getToken` method, this means that only a single thread can try to get a token, which makes sense since we'll be computing the available tokens in a critical section.

We need to think about the following three cases to roll out our algorithm. Let's assume the maximum allowed tokens our bucket can hold is 5.

- The last request for token was more than 5 seconds ago: In this scenario, each elapsed second would have generated one token which may total more than five tokens since the last request was more than 5 seconds ago. We simply need to set the maximum tokens available to 5 since that is the most the bucket will hold and return one token out of those 5.
- The last request for token was within a window of 5 seconds: In this scenario, we need to calculate the new tokens generated since the last request and add them to the unused tokens we already have. We then return 1 token from the count.
- The last request was within a 5-second window and all the tokens are used up: In this scenario, there's no option but to sleep for a whole second to guarantee that a token would become available and then let the thread return. While we `sleep()`, the monitor would still be held by the token-requesting thread and any new threads invoking `getToken` would get blocked, waiting for the monitor to become available.

The above logic is translated into code below

```

public class TokenBucketFilter {

    private int MAX_TOKENS;
    private long lastRequestTime = System.currentTimeMillis();
    long possibleTokens = 0;

    public TokenBucketFilter(int maxTokens) {
        MAX_TOKENS = maxTokens;
    }
}

```

```

    }

    synchronized void getToken() throws InterruptedException {

        // Divide by a 1000 to get granularity at the second level.
        possibleTokens += (System.currentTimeMillis() - lastRequestTime
) / 1000;

        if (possibleTokens > MAX_TOKENS) {
            possibleTokens = MAX_TOKENS;
        }

        if (possibleTokens == 0) {
            Thread.sleep(1000);
        } else {
            possibleTokens--;
        }
        lastRequestTime = System.currentTimeMillis();

        System.out.println(
            "Granting " + Thread.currentThread().getName() + " token a
t " + (System.currentTimeMillis() / 1000));
    }
}

```

You can see the final solution comes out to be very trivial without the requirement for creating a bucket-filling thread of sorts, that runs perpetually and increments a counter every second to reflect the addition of a token to the bucket. Many candidates initially get off-track by taking this approach. Though you might be able to solve this problem using the mentioned approach, the code would unnecessarily be complex and unwieldy.

Note we achieve thread-safety by simply adding synchronized to the `getToken` method. We can have finer grained synchronization inside the method, but that wouldn't help since the entire code snippet within the method is critical and would be guarded by a lock.

If you execute the code below, you'll see we create a token bucket with max tokens set to 1 and have ten threads request for a token. The threads are shown being granted tokens at exactly 1-second intervals instead of all at once. The program output displays the timestamps at which each thread gets the token and we can verify the timestamps are 1 second apart.

Complete Code#

Below is the complete code for the problem:

```

import java.util.HashSet;
import java.util.Set;

```

```

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        TokenBucketFilter.runTestMaxTokenIs1();
    }
}

class TokenBucketFilter {

    private int MAX_TOKENS;
    private long lastRequestTime = System.currentTimeMillis();
    long possibleTokens = 0;

    public TokenBucketFilter(int maxTokens) {
        MAX_TOKENS = maxTokens;
    }

    synchronized void getToken() throws InterruptedException {

        possibleTokens += (System.currentTimeMillis() - lastRequestTime) / 1000;

        if (possibleTokens > MAX_TOKENS) {
            possibleTokens = MAX_TOKENS;
        }

        if (possibleTokens == 0) {
            Thread.sleep(1000);
        } else {
            possibleTokens--;
        }
        lastRequestTime = System.currentTimeMillis();

        System.out.println("Granting " + Thread.currentThread().getName() + " token at " +
                           (System.currentTimeMillis() / 1000));
    }

    public static void runTestMaxTokenIs1() throws InterruptedException {

        Set<Thread> allThreads = new HashSet<Thread>();
        final TokenBucketFilter tokenBucketFilter = new TokenBucketFilter(1);

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

            Thread thread = new Thread(new Runnable() {
                public void run() {
                    try {
                        tokenBucketFilter.getToken();
                    } catch (InterruptedException ie) {
                        System.out.println("We have a problem");
                    }
                }
            });
            allThreads.add(thread);
            thread.start();
        }

        for (Thread thread : allThreads) {
            try {
                thread.join();
            } catch (InterruptedException ie) {
                System.out.println("We have a problem");
            }
        }
    }
}

```

```

        }
    }
});

thread.setName("Thread_" + (i + 1));
allThreads.add(thread);
}

for (Thread t : allThreads) {
    t.start();
}

for (Thread t : allThreads) {
    t.join();
}
}
}
}

```

Below is a more involved test where we let the token bucket filter object receive no token requests for the first **10** seconds.

```

import java.util.HashSet;
import java.util.Set;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        TokenBucketFilter.runTestMaxTokenIsTen();
    }
}

class TokenBucketFilter {

    private int MAX_TOKENS;
    private long lastRequestTime = System.currentTimeMillis();
    long possibleTokens = 0;

    public TokenBucketFilter(int maxTokens) {
        MAX_TOKENS = maxTokens;
    }

    synchronized void getToken() throws InterruptedException {

        possibleTokens += (System.currentTimeMillis() - lastRequestTime) / 1000;

        if (possibleTokens > MAX_TOKENS) {
            possibleTokens = MAX_TOKENS;
        }

        if (possibleTokens == 0) {
            Thread.sleep(1000);
        } else {
    
```

```

        possibleTokens--;
    }
    lastRequestTime = System.currentTimeMillis();

    System.out.println("Granting " + Thread.currentThread().getName() + " token at " +
    (System.currentTimeMillis() / 1000));
}

public static void runTestMaxTokenIsTen() throws InterruptedException {

    Set<Thread> allThreads = new HashSet<Thread>();
    final TokenBucketFilter tokenBucketFilter = new TokenBucketFilter(5);

    // Sleep for 10 seconds.
    Thread.sleep(10000);

    // Generate 12 threads requesting tokens almost all at once.
    for (int i = 0; i < 12; i++) {

        Thread thread = new Thread(new Runnable() {
            public void run() {
                try {
                    tokenBucketFilter.getToken();
                } catch (InterruptedException ie) {
                    System.out.println("We have a problem");
                }
            }
        });
        thread.setName("Thread_" + (i + 1));
        allThreads.add(thread);
    }

    for (Thread t : allThreads) {
        t.start();
    }

    for (Thread t : allThreads) {
        t.join();
    }
}
}

```

The output will show that the first five threads are granted tokens immediately at the same second granularity instant. After that, the subsequent threads are slowly given tokens at an interval of 1 second since one token gets generated every second.

The astute reader would have noticed a problem or a deficiency in our solution. We wait an entire 1 second before we let a thread return with a token. Is that correct? Say we were 20 milliseconds away from getting the next token, but we ended up waiting a full 1000 milliseconds before declaring we have a token available. We can eliminate this inefficiency by maintaining more state however for an interview problem the given solution is sufficient.

Follow-up Questions#

For the brave, we recommend implementing the following challenges.

1. Grant tokens to threads in a FIFO order
2. Generalize the solution for any rate of token generation

... continued

This lesson explains how to solve the token bucket filter problem using threads.

Solution using a background thread#

The previous solution consisted of manipulating pointers in time, thus avoiding threads altogether. Another solution is to use threads to solve the token bucket filter problem. We instantiate one thread to add a token to the bucket after every one second. The user thread invokes the `getToken()` method and is granted one if available.

One simplification as a result of using threads is we now only need to remember the current number of tokens held by the token bucket filter object. We'll add an additional method `daemonThread()` that will be executed by the thread that adds a token every second to the bucket. The skeleton of our class looks as follows:

```
public class MultithreadedTokenBucketFilter {  
    private long possibleTokens = 0;  
    private final int MAX_TOKENS;  
    private final int ONE_SECOND = 1000;  
  
    public MultithreadedTokenBucketFilter(int maxTokens) {  
        MAX_TOKENS = maxTokens;  
    }  
  
    private void daemonThread() {
```

```
}

    void getToken() throws InterruptedException
    {
    }

}
```

The logic of the daemon thread is simple. It sleeps for one second, wakes up, checks if the number of tokens in the bucket is less than the maximum allowed tokens, if yes increments the `possibleTokens` variable and if not goes back to sleep for a second again.

The implementation of the `getToken()` is even simpler. The user thread checks if the number of tokens is greater than zero, if yes it simulates taking away a token by decrementing the variable `possibleTokens`. If the number of available tokens is zero then the user thread must wait and be notified only when the daemon thread has added a token. We can use the current token bucket object `this` to wait and notify. The implementation of the `getToken()` method is shown below:

```
void getToken() throws InterruptedException {

    synchronized (this) {
        while (possibleTokens == 0) {
            this.wait();
        }
        possibleTokens--;
    }

    System.out.println(
        "Granting " + Thread.currentThread().getName() + " token at " +
        System.currentTimeMillis() / 1000);
}
```

Note that we are manipulating the shared mutable object `possibleTokens` in a synchronized block. Additionally, we `wait()` when the number of tokens is zero. The implementation of the daemon thread is given below. It runs in a perpetual loop.

```
private void daemonThread() {

    while (true) {
        synchronized (this) {
            if (possibleTokens < MAX_TOKENS) {
                possibleTokens++;
            }
            this.notify();
        }
    }
}
```

```
    }
    try {
        Thread.sleep(ONE_SECOND);
    } catch (InterruptedException ie) {
        // swallow exception
    }
}
```

The complete implementation along with a test-case appears in the code widget below:

```
import java.util.HashSet;
import java.util.Set;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        Set<Thread> allThreads = new HashSet<Thread>();
        final MultithreadedTokenBucketFilter tokenBucketFilter = new
MultithreadedTokenBucketFilter(1);

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

            Thread thread = new Thread(new Runnable() {

                public void run() {
                    try {
                        tokenBucketFilter.getToken();
                    } catch (InterruptedException ie) {
                        System.out.println("We have a problem");
                    }
                }
            });
            thread.setName("Thread_" + (i + 1));
            allThreads.add(thread);
        }

        for (Thread t : allThreads) {
            t.start();
        }

        for (Thread t : allThreads) {
            t.join();
        }

    }
}

class MultithreadedTokenBucketFilter {
```

```

private long possibleTokens = 0;
private final int MAX_TOKENS;
private final int ONE_SECOND = 1000;

public MultithreadedTokenBucketFilter(int maxTokens) {

    MAX_TOKENS = maxTokens;

    // Never start a thread in a constructor
    Thread dt = new Thread(() -> {
        daemonThread();
    });
    dt.setDaemon(true);
    dt.start();
}

private void daemonThread() {

    while (true) {

        synchronized (this) {
            if (possibleTokens < MAX_TOKENS) {
                possibleTokens++;
            }
            this.notify();
        }

        try {
            Thread.sleep(ONE_SECOND);
        } catch (InterruptedException ie) {
            // swallow exception
        }
    }
}

void getToken() throws InterruptedException {

    synchronized (this) {
        while (possibleTokens == 0) {
            this.wait();
        }
        possibleTokens--;
    }

    System.out.println(
        "Granting " + Thread.currentThread().getName() + " token at " +
        System.currentTimeMillis() / 1000);
}

```

```
}
```

We reuse the test-case from the previous lesson, where we create a token bucket with max tokens set to 1 and have ten threads request for a token. The threads are shown being granted tokens at exactly 1-second intervals instead of all at once. The program output displays the timestamps at which each thread gets the token and we can verify the timestamps are 1 second apart. Additionally, we mark the daemon thread as background so that it exits when the application terminates.

Using a factory#

The problem with the above solution is that we start our thread in the constructor. Never start a thread in a constructor as the child thread can attempt to use the not-yet-fully constructed object using `this`. This is an anti-pattern. Some candidates present this solution when attempting to solve token bucket filter problem using threads. However, when checked, few candidates can reason why starting threads in a constructor is a bad choice.

There are two ways to overcome this problem, the naive but correct solution is to start the daemon thread outside of the `MultithreadedTokenBucketFilter` object. However, the con of this approach is that the management of the daemon thread spills outside the class. Ideally, we want the class to encapsulate all the operations related with the management of the token bucket filter and only expose the public API to the consumers of our class, as per good object orientated design. This situation is a great for using the **Simple Factory** design pattern. We'll create a factory class which produces token bucket filter objects and also starts the daemon thread only when the object is full constructed. If you are unaware of this pattern, I'll take the liberty insert a shameless marketing plug here and refer you to this [design patterns](#) course to get up to speed.

Our token bucket filter factory will expose a method `makeTokenBucketFilter()` that will return an object of type token bucket filter. Before returning the object we'll start the daemon thread. Additionally, we don't want consumers to be able to instantiate the token bucket filter objects without interacting with the factory. For this reason, we'll make the class `MultithreadedTokenBucketFilter` private and nest it within the factory class. We'll also add an abstract `TokenBucketFilter` class that consumers can use to reference the object returned from our `makeTokenBucketFilter()` method. The class `MultithreadedTokenBucketFilter` will extend the abstract class `TokenBucketFilter`.

The complete code with the same test case appears below.

Main.java

```
import java.util.HashSet;
import java.util.Set;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        Set<Thread> allThreads = new HashSet<Thread>();
        TokenBucketFilter tokenBucketFilter =
            TokenBucketFilterFactory.makeTokenBucketFilter(1);

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

            Thread thread = new Thread(new Runnable() {

                public void run() {
                    try {
                        tokenBucketFilter.getToken();
                    } catch (InterruptedException ie) {
                        System.out.println("We have a problem");
                    }
                }
            });
            thread.setName("Thread_" + (i + 1));
            allThreads.add(thread);
        }

        for (Thread t : allThreads) {
            t.start();
        }

        for (Thread t : allThreads) {
            t.join();
        }

    }
}
```

TokenBucketFilter.java

```
public abstract class TokenBucketFilter {
    public void getToken() throws InterruptedException {
    }
}
```

TokenBucketFilterFactory.java

```
public final class TokenBucketFilterFactory {  
  
    // Force users to interact with the factory  
    // only through the static methods  
    private TokenBucketFilterFactory() {  
    }  
  
    static public TokenBucketFilter makeTokenBucketFilter(int capacity) {  
        MultithreadedTokenBucketFilter tbf = new MultithreadedTokenBucketFilter(capacity);  
        tbf.initialize();  
        return tbf;  
    }  
  
    private static class MultithreadedTokenBucketFilter extends TokenBucketFilter {  
        private long possibleTokens = 0;  
        private final int MAX_TOKENS;  
        private final int ONE_SECOND = 1000;  
  
        // MultithreadedTokenBucketFilter object can only  
        MultithreadedTokenBucketFilter(int maxTokens) {  
            MAX_TOKENS = maxTokens;  
        }  
  
        void initialize() {  
            // Never start a thread in a constructor  
            Thread dt = new Thread(() -> {  
                daemonThread();  
            });  
            dt.setDaemon(true);  
            dt.start();  
        }  
  
        private void daemonThread() {  
  
            while (true) {  
                synchronized (this) {  
                    if (possibleTokens < MAX_TOKENS) {  
                        possibleTokens++;  
                    }  
                    this.notify();  
                }  
                try {  
                    Thread.sleep(ONE_SECOND);  
                } catch (InterruptedException ie) {  
                    // swallow exception  
                }  
            }  
        }  
    }  
}
```

```

    }

    public void getToken() throws InterruptedException {
        synchronized (this) {
            while (possibleTokens == 0) {
                this.wait();
            }
            possibleTokens--;
        }

        System.out.println(
            "Granting " + Thread.currentThread().getName() + " token at " +
        System.currentTimeMillis() / 1000);
    }
}
}

```

Thread Safe Deferred Callback

Asynchronous programming involves being able to execute functions at a future occurrence of some event. Designing a thread-safe deferred callback class becomes a challenging interview question.

Problem Statement#

Design and implement a thread-safe class that allows registration of callback methods that are executed after a user specified time interval in seconds has elapsed.

Solution#

Let us try to understand the problem without thinking about concurrency. Let's say our class exposes an API called `registerCallback()` that'll take a parameter of type `Callback`, which we'll define later. Anyone calling this API should be able to specify after how many seconds should our executor invoke the passed in callback.

One naive way to solve this problem is to have a busy thread that continuously loops over the list of callbacks and executes them as they become due. However, the challenge here is to design a solution which doesn't involve a busy thread.

If we restrict ourselves to use only concurrency constructs offered by Java then one possible solution is to have an execution thread that maintains a

priority queue of callbacks ordered by the time remaining to execute each of the callbacks. The execution thread can sleep for the duration equal to the time duration before the earliest callback in the min-heap becomes due for execution.

Consumer threads can come and add their desired callbacks in the min-heap within a critical section. However, whenever a consumer thread requests a callback be registered, the caveat is to wake up the execution thread and recalculate the minimum duration it needs to sleep for before the earliest callback becomes due for execution. It is possible that a callback with an earlier due timestamp gets added by a consumer thread while the executor thread is currently asleep for a duration, calculated for a callback due later than the one just added.

Consider this example: initially, the execution thread is sleeping for 30 mins before any callback in the min-heap is due. A consumer thread comes along and adds a callback to be executed after 5 minutes. The execution thread would need to wake up and reset itself to sleep for only 5 minutes instead of 30 minutes. Once we find an elegant way of capturing this logic our problem is pretty much solved.

Let's see how the skeleton of our class would look like:

Class Skeleton#

```
public class DeferredCallbackExecutor {  
    PriorityQueue<CallBack> q = new PriorityQueue<CallBack>(new Comparator  
<CallBack>() {  
        public int compare(CallBack o1, CallBack o2) {  
            return (int) (o1.executeAt - o2.executeAt);  
        }  
    });  
  
    // Run by the Executor Thread  
    public void start() throws InterruptedException {  
    }  
  
    // Called by Consumer Threads to register callback  
    public void registerCallback(CallBack callBack) {  
    }  
  
    /**  
     * Represents the class which holds the callback. For simplicity instead of  
     * executing a method, we print a message.  
     */  
    static class CallBack {  
    }
```

```

    long executeAt;
    String message;

    public CallBack(long executeAfter, String message) {
        this.executeAt = System.currentTimeMillis() + executeAfter * 1
000;
        this.message = message;
    }
}

```

We define a simple `CallBack` class, an object of which will be passed into the `registerCallback()` method. This method will add a new callback to our min heap. In Java the generic `PriorityQueue` is an implementation of a heap which can be passed a comparator to either act as a min or max heap. In our case, we pass in a comparator in the constructor so that the callbacks are ordered by their execution times, the earliest callback to be executed sits at the top of the heap.

For guarding access to critical sections we'll use an object of the `ReentrantLock` class offered by Java. It acts similar to a mutex. Also we'll introduce the use of a Condition variable. The execution thread will wait on it while the consumer threads will signal it. The condition variable allows the consumer threads to wake up the execution thread whenever a new callback is registered. Let's write out what we just discussed as code.

```

public class DeferredCallbackExecutor {

    PriorityQueue<CallBack> q = new PriorityQueue<CallBack>(new Comparator
<CallBack>() {

        public int compare(CallBack o1, CallBack o2) {
            return (int) (o1.executeAt - o2.executeAt);
        }
    });
    // Lock to guard critical sections
    ReentrantLock lock = new ReentrantLock();

    // Condition to make execution thread wait on
    Condition newCallbackArrived = lock.newCondition();

    public void start() throws InterruptedException {
    }

    public void registerCallback(CallBack callBack) {
        lock.lock();
        q.add(callBack);
        newCallbackArrived.signal();
        lock.unlock();
    }
}

```

```

static class CallBack {

    long executeAt;
    String message;

    public CallBack(long executeAfter, String message) {
        this.executeAt = System.currentTimeMillis() + executeAfter * 1
000;
        this.message = message;
    }
}

```

Note how in `registerCallback()` method we lock the critical section before adding the callback to the queue. Also, we signal the condition associated with the lock. As a reminder, note the execution thread, if waiting on the condition variable, will not be able to make progress until the consumer thread gives up the lock, even though the condition has been signaled.

Now lets come to the meat of our solution which is to design the execution thread's workflow. The thread will run the `start()` method and enter into a perpetual loop. The flow will be as follows:

- Initially the queue will be empty and the execution thread should just wait indefinitely on the condition variable to be signaled.
- When the first callback gets registered, we note how many seconds after its arrival does it need to be executed and `await()` on the condition variable for that many seconds.
- Now two things are possible at this point. No new callbacks arrive, in which case the executor thread completes waiting and polls the queue for tasks that should be executed and starts executing them.

Or that another callback arrives, in which case the consumer thread will signal the condition variable `newCallbackArrived` to wake up the execution thread and have it re-evaluate the duration it can sleep for before the earliest callback becomes due.

This flow is caputed in the code below:

```

private long findSleepDuration() {
    long currentTime = System.currentTimeMillis();
    return q.peek().executeAt - currentTime;
}

public void start() throws InterruptedException {

```

```

long sleepFor = 0;

while (true) {
    // lock the critical section
    lock.lock();

    // if no item in the queue, wait indefinitely for one to arrive
    while (q.size() == 0) {
        newCallbackArrived.await();
    }

    // loop till all callbacks have been executed
    while (q.size() != 0) {

        // find the minimum time execution thread should
        // sleep for before the next callback becomes due
        sleepFor = findSleepDuration();

        // If the callback is due break from loop and start
        // executing the callback
        if(sleepFor <=0)
            break;

        // sleep until the earliest due callback can be executed
        newCallbackArrived.await(sleepFor, TimeUnit.MILLISECONDS);
    }

    // Because we have a min-heap the first element of the queue
    // is necessarily the one which is due.
    CallBack cb = q.poll();
    System.out.println(
        "Executed at " + System.currentTimeMillis() / 1000 + " required at " + cb.executeAt /
1000
        + ": message:" + cb.message);

    // Don't forget to unlock the critical section
    lock.unlock();
}
}

```

The working of the above snippet is explained below

- Initially, the queue is empty and the executor thread will simply `await()` indefinitely on the condition `newCallbackArrived` to be signaled. Note we wrap the waiting in a while loop to cater for spurious wakeups.

- If the queue is not empty, say if the executor thread is created later than the consumer threads, then the executor will fall into the second while loop and either wait for the callback to become due or if one is already due break out of the while loop and execute the due callback.
- For all other happy path cases, adding a callback to the queue will always signal the awaiting executor thread to wake up and recalculate the time it needs to sleep before the next callback is ready to be executed.
- Note that both the `await()` calls are properly enclosed by while loops to cater for spurious wakeups. In the second while loop, if a spurious wakeup happens, the executor thread recalculates the sleep time, find it to be greater than zero and goes back to sleeping until a callback becomes due.

Complete Code#

The complete code with the test case appears below. We insert ten callbacks sequentially waiting randomly between insertions. The output shows the epoch seconds at which the callback was expected to be executed and the actual time at which it got executed. Both of these values, for all the callbacks, should be the same or differ very slightly to account for the minuscule time it for the executor thread to wake up and execute the callback.

The code includes a test case where ten callbacks are executed. Since the code runs in the browser, it might timeout before printing the complete output.

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

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        DeferredCallbackExecutor.runTestTenCallbacks();
    }
}

class DeferredCallbackExecutor {

    PriorityQueue<CallBack> q = new PriorityQueue<CallBack>(new Comparator<CallBack>() {
        public int compare(CallBack o1, CallBack o2) {
            return (int) (o1.executeAt - o2.executeAt);
        }
    })
}
```

```

    });
    ReentrantLock lock = new ReentrantLock();
    Condition newCallbackArrived = lock.newCondition();

    private long findSleepDuration() {
        long currentTime = System.currentTimeMillis();
        return q.peek().executeAt - currentTime;
    }

    public void start() throws InterruptedException {
        long sleepFor = 0;

        while (true) {

            lock.lock();

            while (q.size() == 0) {
                newCallbackArrived.await();
            }

            while (q.size() != 0) {
                sleepFor = findSleepDuration();

                if(sleepFor <=0)
                    break;

                newCallbackArrived.await(sleepFor, TimeUnit.MILLISECONDS);
            }

            CallBack cb = q.poll();
            System.out.println(
                "Executed at " + System.currentTimeMillis()/1000 + " required at " +
                cb.executeAt/1000
                + ": message:" + cb.message);

            lock.unlock();
        }
    }

    public void registerCallback(CallBack callBack) {
        lock.lock();
        q.add(callBack);
        newCallbackArrived.signal();
        lock.unlock();
    }

    static class CallBack {
        long executeAt;

```

```

String message;

public CallBack(long executeAfter, String message) {
    this.executeAt = System.currentTimeMillis() + (executeAfter * 1000);
    this.message = message;
}
}

public static void runTestTenCallbacks() throws InterruptedException {
    Set<Thread> allThreads = new HashSet<Thread>();
    final DeferredCallbackExecutor deferredCallbackExecutor = new
DeferredCallbackExecutor();

    Thread service = new Thread(new Runnable() {
        public void run() {
            try {
                deferredCallbackExecutor.start();
            } catch (InterruptedException ie) {

            }
        }
    });
}

service.start();

for (int i = 0; i < 10; i++) {
    Thread thread = new Thread(new Runnable() {
        public void run() {
            CallBack cb = new CallBack(1, "Hello this is " + Thread.currentThread().getName());
            deferredCallbackExecutor.registerCallback(cb);
        }
    });
    thread.setName("Thread_" + (i + 1));
    thread.start();
    allThreads.add(thread);
    Thread.sleep(1000);
}

for (Thread t : allThreads) {
    t.join();
}
}
}

```

Here's another test-case, which first submits a callback that should get executed after eight seconds. Three seconds later another call back is submitted which should be executed after only one second. The callback being submitted later should execute first. The test run would timeout if

run in the browser since the callback service is a perpetual thread but from the output you can observe the callback submitted second execute first.

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

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        DeferredCallbackExecutor.runLateThenEarlyCallback();
    }
}

class DeferredCallbackExecutor {

    private static Random random = new Random(System.currentTimeMillis());

    PriorityQueue<CallBack> q = new PriorityQueue<CallBack>(new Comparator<CallBack>() {
        public int compare(CallBack o1, CallBack o2) {
            return (int) (o1.executeAt - o2.executeAt);
        }
    });
    ReentrantLock lock = new ReentrantLock();
    Condition newCallbackArrived = lock.newCondition();

    private long findSleepDuration() {
        long currentTime = System.currentTimeMillis();
        return q.peek().executeAt - currentTime;
    }

    public void start() throws InterruptedException {
        long sleepFor = 0;

        while (true) {

            lock.lock();

            while (q.size() == 0) {
                newCallbackArrived.await();
            }

            while (q.size() != 0) {
                sleepFor = findSleepDuration();

                if(sleepFor <=0)
                    break;

                newCallbackArrived.await(sleepFor, TimeUnit.MILLISECONDS);
            }
        }
    }
}
```

```

    }

    CallBack cb = q.poll();
    System.out.println(
        "Executed at " + System.currentTimeMillis()/1000 + " required at " +
    cb.executeAt/1000
        + ": message:" + cb.message);

    lock.unlock();
}
}

public void registerCallback(CallBack callBack) {
    lock.lock();
    q.add(callBack);
    newCallbackArrived.signal();
    lock.unlock();
}

static class CallBack {
    long executeAt;
    String message;

    public CallBack(long executeAfter, String message) {
        this.executeAt = System.currentTimeMillis() + executeAfter * 1000;
        this.message = message;
    }
}

public static void runLateThenEarlyCallback() throws InterruptedException {
    final DeferredCallbackExecutor deferredCallbackExecutor = new
DeferredCallbackExecutor();

    Thread service = new Thread(new Runnable() {
        public void run() {
            try {
                deferredCallbackExecutor.start();
            } catch (InterruptedException ie) {
            }
        }
    });
}

service.start();

Thread lateThread = new Thread(new Runnable() {
    public void run() {
        CallBack cb = new CallBack(8, "Hello this is the callback submitted first");
        deferredCallbackExecutor.registerCallback(cb);
    }
});

```

```

        }
    });
lateThread.start();

Thread.sleep(3000);

Thread earlyThread = new Thread(new Runnable() {
    public void run() {
        CallBack cb = new CallBack(1, "Hello this is callback submitted second");
        deferredCallbackExecutor.registerCallback(cb);
    }
});
earlyThread.start();

lateThread.join();
earlyThread.join();
}
}

```

Implementing Semaphore

Learn how to design and implement a simple semaphore class in Java.

Problem Statement#

Java does provide its own implementation of Semaphore, however, Java's semaphore is initialized with an initial number of permits, rather than the maximum possible permits and the developer is expected to take care of always releasing the intended number of maximum permits.

Briefly, a semaphore is a construct that allows some threads to access a fixed set of resources in parallel. Always think of a semaphore as having a fixed number of permits to give out. Once all the permits are given out, requesting threads, need to wait for a permit to be returned before proceeding forward.

Your task is to implement a semaphore which takes in its constructor the maximum number of permits allowed and is also initialized with the same number of permits.

Solution#

Given the above definition we can now start to think of what functions our Semaphore class will need to expose. We need a function to "gain the permit" and a function to "return the permit".

1. `acquire()` function to simulate gaining a permit
2. `release()` function to simulate releasing a permit

The constructor accepts an integer parameter defining the number of permits available with the semaphore. Internally we need to store a count which keeps track of the permits given out so far.

The skeleton for our Semaphore class looks something like this so far.

```
public class CountingSemaphore {  
  
    int usedPermits = 0; // permits given out  
    int maxCount; // max permits to give out  
  
    public CountingSemaphore(int count) {  
        this.maxCount = count;  
    }  
  
    public synchronized void acquire() throws InterruptedException {  
    }  
  
    public synchronized void release() throws InterruptedException {  
    }  
}
```

Note we have added the `synchronized` keyword to both the class methods. Adding the synchronized keyword causes only a single thread to execute either of the methods. If a thread is currently executing `acquire()` then another thread can't execute `release()` on the same semaphore object.

Note this will guarantee that the `usedPermits` variable is correctly incremented or decremented.

The astute observer would question why we don't take the locking to finer grained level and use java's lock so that multiple threads can call either of the two functions. With `synchronized` only one thread can call either release or acquire. However, the counter to that is, even with finer grained locking the entire code blocks within the two methods will be guarded by a single lock and that would pretty much be the same as putting synchronized on the methods definitions.

Now let us fill in the implementation for our acquire method. When can a thread not be allowed to acquire a semaphore? When all the permits are out! This implies we'll need to `wait()` when `usedPermits == maxCount` If this condition isn't true we simply increment `usedPermits` to simulate giving out a permit.

The implementation of the acquire method appears below. Note that we are also `notify()`-ing at the end of the method. We'll talk shortly, about why we need it.

```
1. public class CountingSemaphore {
2.
3.     int usedPermits = 0; // permits given out
4.     int maxCount; // max permits to give out
5.
6.     public CountingSemaphore(int count) {
7.         this.maxCount = count;
8.     }
9.
10.    public synchronized void acquire() throws InterruptedException {
11.
12.        while (usedPermits == maxCount)
13.            wait();
14.
15.        usedPermits++;
16.        notify();
17.    }
18.
19.    public synchronized void release() throws InterruptedException {
20.    }
21. }
```

Implementing the release method should require a simple decrement of the `usedPermits` variable. However when should we block a thread from proceeding forward like we did in `acquire()` method? If `usedPermits == 0` then it won't make sense to decrement `usedPermits` and we should block at this condition.

This might seem counter-intuitive, you might ask why would someone call `release()` before calling `acquire()` - This is entirely possible since semaphore can also be used for signalling between threads. A thread can call `release()` on a semaphore object before another thread calls `acquire()` on the same semaphore object. There is no concept of ownership for a semaphore ! Hence different threads can call acquire or release methods as they deem fit.

This also means that whenever we decrement or increment the `usedPermits` variable we need to call `notify()` so that any waiting thread in the other method is able to move forward. The full implementation appears below

```
1. public class CountingSemaphore {
2.
3.     int usedPermits = 0; // permits given out
4.     int maxCount; // max permits to give out
```

```

5.     public CountingSemaphore(int count) {
6.         this.maxCount = count;
7.     }
8.
9.     }
10.
11.    public synchronized void acquire() throws InterruptedException {
12.
13.        while (usedPermits == maxCount)
14.            wait();
15.
16.        usedPermits++;
17.        notify();
18.    }
19.
20.    public synchronized void release() throws InterruptedException {
21.
22.        while (usedPermits == 0)
23.            wait();
24.
25.        usedPermits--;
26.        notify();
27.    }
28. }
```

As a follow-up, notice that we increment/decrement the `usedPermits` variable on **line 16** and **25** respectively and then call `notify()`. Does it matter if we switch the order of the two statements, i.e. call `notify()` first and then manipulate `usedPermits`? The answer is no.

When `notify()` is called, the executing thread is still synchronized on the semaphore object and any other thread will not be scheduled until the executing thread exists the `release()` or `acquire()` method and by then the `usedPermits` variable has already been incremented or decremented even though that is the last statement in each method.

Also, remember that semaphores solve the problem of missed signals between cooperating threads. Imagine a producer/consumer application where the producer wants to notify the consumer of available content for consumption. The producer can call `acquire()` on a binary semaphore (one with `max permits = 1`) and the consumer can call `release()` before attempting to consume. This way the “signal” is in a sense stored for the consumer whenever the producer produces something.

Complete Code#

The complete code appears below along with a test. Note how we acquire and release the semaphore in different threads in different methods, something not possible with a mutex. Thread t1 always acquires the semaphore while thread t2 always releases it. The semaphore has a max permit of 1 so you'll see the output interleaved between the two threads.

You might see the print statements from the two threads not interleave each other and may appear twice in succession. This is possible because of how threads get scheduled for execution and also because we start with an unused permit.

The astute reader would also observe that the given solution will always block if the semaphore is initialized with zero permits

```
class Demonstration {  
    public static void main( String args[] ) throws InterruptedException {  
  
        final CountingSemaphore cs = new CountingSemaphore(1);  
  
        Thread t1 = new Thread(new Runnable() {  
  
            @Override  
            public void run() {  
                try {  
                    for (int i = 0; i < 5; i++) {  
                        cs.acquire();  
                        System.out.println("Ping " + i);  
                    }  
                } catch (InterruptedException ie) {  
  
                }  
            }  
        });  
  
        Thread t2 = new Thread(new Runnable() {  
  
            @Override  
            public void run() {  
                for (int i = 0; i < 5; i++) {  
                    try {  
                        cs.release();  
                        System.out.println("Pong " + i);  
                    } catch (InterruptedException ie) {  
  
                    }  
                }  
            }  
        });  
  
        t2.start();  
        t1.start();  
        t1.join();  
        t2.join();  
    }  
}
```

```

}

class CountingSemaphore {

    int usedPermits = 0;
    int maxCount;

    public CountingSemaphore(int count) {
        this.maxCount = count;
    }

    public synchronized void acquire() throws InterruptedException {
        while (usedPermits == maxCount)
            wait();

        notify();
        usedPermits++;
    }

    public synchronized void release() throws InterruptedException {
        while (usedPermits == 0)
            wait();

        usedPermits--;
        notify();
    }
}

```

ReadWrite Lock

We discuss a common interview question involving synchronization of multiple reader threads and a single writer thread.

Problem Statement#

Imagine you have an application where you have multiple readers and multiple writers. You are asked to design a lock which lets multiple readers read at the same time, but only one writer write at a time.

Solution#

First of all let us define the APIs our class will expose. We'll need two for writer and two for reader. These are:

- acquireReadLock
- releaseReadLock
- acquireWriteLock
- releaseWriteLock

This problem becomes simple if you think about each case:

1. Before we allow a reader to enter the critical section, we need to make sure that there's no **writer** in progress. It is ok to have other readers in the critical section since they aren't making any modifications
2. Before we allow a writer to enter the critical section, we need to make sure that there's **no reader or writer** in the critical section.

```
public class ReadWriteLock {  
    public synchronized void acquireReadLock() throws InterruptedException  
    {  
    }  
  
    public synchronized void releaseReadLock()  
    {  
    }  
  
    public synchronized void acquireWriteLock() throws InterruptedException  
    {  
    }  
  
    public synchronized void releaseWriteLock()  
    {  
    }  
}
```

Note that all the methods are synchronized on the ReadWriteLock object itself.

Let's start with the reader use case. We can have multiple readers acquire the read lock and to keep track of all of them; we'll need a count. We increment this count whenever a reader acquires a read lock and decrement it whenever a reader releases it.

Releasing the read lock is easy but before we acquire the read lock, we need to be sure that no other writer is currently writing. Again, we'll need some variable to keep track of whether a writer is writing. Since only a single writer can write at a given point in time, we can just keep a boolean variable to denote if the write lock is acquired or not. Let's translate what we have discussed so far into code.

```

public class ReadWriteLock {

    boolean isWriteLocked = false;
    int readers = 0;

    public synchronized void acquireReadLock() throws InterruptedException {
        while (isWriteLocked) {
            wait();
        }
        readers++;
    }

    public synchronized void releaseReadLock() {
        readers--;
    }

    public synchronized void acquireWriteLock() throws InterruptedException {
    }

    public synchronized void releaseWriteLock() {
    }
}

```

Note how we are checking in a loop whether `isWriteLocked` is true and then calling `wait()`. Also pay attention to the fact that the methods are synchronized so only one reader will be able to decrement reader in `releaseReadLock`.

For the writer case, releasing the lock would be as simple as setting the `isWriteLocked` variable to false but don't forget to call `notify()` too since there might be readers waiting in the `acquireReadLock()` method.

Acquiring the write lock is a little tricky, we have to check two things whether any other writer has already set `isWriteLocked` to true and also if any reader has incremented the `readers` variable. If `isWriteLocked` equals false and no reader is writing then the writer should proceed forward.

```

public class ReadWriteLock {

    boolean isWriteLocked = false;
    int readers = 0;

    public synchronized void acquireReadLock() throws InterruptedException {
        while (isWriteLocked) {
            wait();
        }
        readers++;
    }

    public synchronized void releaseReadLock() {
        readers--;
    }

    public synchronized void acquireWriteLock() throws InterruptedException {
        if (!isWriteLocked && readers == 0) {
            isWriteLocked = true;
            notify();
        }
    }

    public synchronized void releaseWriteLock() {
        if (isWriteLocked) {
            isWriteLocked = false;
            notifyAll();
        }
    }
}

```

```

    }

    public synchronized void releaseReadLock() {
        readers--;
    }

    public synchronized void acquireWriteLock() throws InterruptedException {
        while (isWriteLocked || readers != 0) {
            wait();
        }
        isWriteLocked = true;
    }

    public synchronized void releaseWriteLock() {
        isWriteLocked = false;
        notify();
    }
}

```

The astute reader will notice a bug in the code we have so far. Try finding it, before reading ahead !

For the impatient, note that in our `acquireWriteLock()` method we have a while loop which has `readers != 0` condition. We should remember to call `notify()` whenever any snippet of code can change the value of the `readers` variable and make the loop condition in `acquireWriteLock()` false. If we don't call `notify` then any thread waiting in the loop will never be woken up.

Within the `releaseReadLock()` method, we should call `notify()` after decrementing readers to make sure that any blocked readers should be able to proceed forward.

```

public class ReadWriteLock {

    boolean isWriteLocked = false;
    int readers = 0;

    public synchronized void acquireReadLock() throws InterruptedException {
        while (isWriteLocked) {
            wait();
        }
        readers++;
    }

    public synchronized void releaseReadLock() {
        readers--;
        notify();
    }
}

```

```

    public synchronized void acquireWriteLock() throws InterruptedException {
        while (isWriteLocked || readers != 0) {
            wait();
        }
        isWriteLocked = true;
    }

    public synchronized void releaseWriteLock() {
        isWriteLocked = false;
        notify();
    }
}

```

Note that with the given implementation, it is possible for a writer to starve and never get a chance to acquire the write lock since there could always be at least one reader which has the read lock acquired.

Complete Code#

The complete code with a test-case appears below. Run the code and examine the output messages. We start a reader and a writer thread initially. The writer blocks until the read lock is released. Also, we release the reader-lock through another reader thread.

A second writer thread is blocked forever since the first writer thread never releases the write-lock. The execution eventually times out.

```

class Demonstration {

    public static void main(String args[]) throws Exception {
        final ReadWriteLock rwl = new ReadWriteLock();

        Thread t1 = new Thread(new Runnable() {

            @Override
            public void run() {
                try {
                    System.out.println("Attempting to acquire write lock in t1: " +
                        System.currentTimeMillis());
                    rwl.acquireWriteLock();
                    System.out.println("write lock acquired t1: " + System.currentTimeMillis());

                    // Simulates write lock being held indefinitely
                    for (;;) {

```

```

        Thread.sleep(500);
    }

} catch (InterruptedException ie) {

}

};

Thread t2 = new Thread(new Runnable() {

@Override
public void run() {
try {

System.out.println("Attempting to acquire write lock in t2: " +
System.currentTimeMillis());
rwL.acquireWriteLock();
System.out.println("write lock acquired t2: " + System.currentTimeMillis());

} catch (InterruptedException ie) {

}

}
});

Thread tReader1 = new Thread(new Runnable() {

@Override
public void run() {
try {

rwL.acquireReadLock();
System.out.println("Read lock acquired: " + System.currentTimeMillis());

} catch (InterruptedException ie) {

}

}
});

Thread tReader2 = new Thread(new Runnable() {

@Override
public void run() {
System.out.println("Read lock about to release: " + System.currentTimeMillis());
rwL.releaseReadLock();
System.out.println("Read lock released: " + System.currentTimeMillis());
}
});

```

```
        }
    });

tReader1.start();
t1.start();
Thread.sleep(3000);
tReader2.start();
Thread.sleep(1000);
t2.start();
tReader1.join();
tReader2.join();
t2.join();
}
}

class ReadWriteLock {

    boolean isWriteLocked = false;
    int readers = 0;

    public synchronized void acquireReadLock() throws InterruptedException {

        while (isWriteLocked) {
            wait();
        }

        readers++;
    }

    public synchronized void releaseReadLock() {
        readers--;
        notify();
    }

    public synchronized void acquireWriteLock() throws InterruptedException {

        while (isWriteLocked || readers != 0) {
            wait();
        }

        isWriteLocked = true;
    }

    public synchronized void releaseWriteLock() {
        isWriteLocked = false;
        notify();
    }
}
```

The write-lock is acquired only after the read-lock is released. If you look at the output, the write lock acquisition timestamp would be between the timestamps of the statements "**read lock about to release**" and "**read lock released**". This is so because timestamps aren't granular enough and read-lock's release timestamp and write-lock's acquisition timestamp might be same.

Also - the read-lock's release statement might get printed after the write-lock's acquisition statement but that is possible if the thread tReader2 gets context-switched as soon as it releases the lock and before it gets a chance to execute the print statement.

Last but not the least, running the above test in the browser would show execution timing out. This is expected as our t1 thread is modelled as a writer thread that never releases the write-lock.

Unisex Bathroom Problem

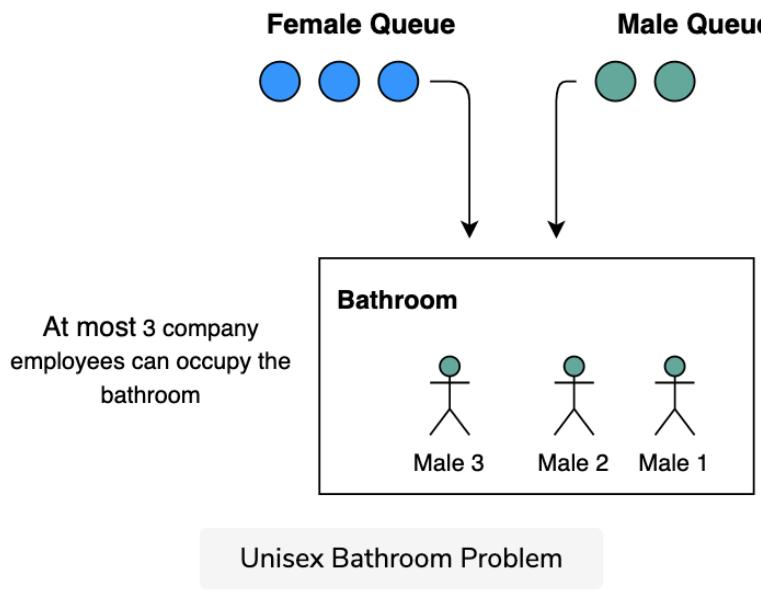
A synchronization practice problem requiring us to synchronize the usage of a single bathroom by both the genders.

Problem Statement#

A bathroom is being designed for the use of both males and females in an office but requires the following constraints to be maintained:

- There cannot be men and women in the bathroom at the same time.
- There should never be more than three employees in the bathroom simultaneously.

The solution should avoid deadlocks. For now, though, don't worry about starvation.



Solution#

First let us come up with the skeleton of our Unisex Bathroom class. We want to model the problem programmatically first. We'll need two APIs, one that is called by a male to use the bathroom and another one that is called by the woman to use the bathroom. Initially our class looks like the following

```
public class UnisexBathroom {

    void maleUseBathroom(String name) throws InterruptedException {
    }

    void femaleUseBathroom(String name) throws InterruptedException {
}
}
```

Let us try to address the first problem of allowing either men or women to use the bathroom. We'll worry about the max employees later. We need to maintain state in a variable which would tell us which gender is currently using the bathroom. Let's call this variable `inUseBy`. To make the code more readable we'll make the type of the variable `inUseBy` a string which can take on the values men, women or none.

We'll also have a method `useBathroom()` that'll mock a person using the bathroom. The implementation of this method will simply sleep the thread using the bathroom for some time.

Assume there's no one in the bathroom and a male thread invokes the `maleUseBathroom()` method, the thread has to check first whether the bathroom is being used by a female thread. If it is indeed being used by a female, then the male thread has to wait for the bathroom to be empty. If the male thread already finds the bathroom empty, which in our scenario it does, the thread simply updates the `inUseBy` variable to "MEN" and proceeds to use the bathroom. After using the bathroom, however, it must let any waiting female threads know that it is done and they can now use the bathroom.

The astute reader would immediately realize that we'll need to guard the variable `inUseBy` since it can possibly be both read and written to by different threads at the same time. Does that mean we should mark our methods as `synchronized`? The wary reader would know that doing so would essentially make the threads serially access the methods, i.e., if one male thread is accessing the bathroom, then another one can't access the bathroom even though the problem says that more than one male should be able to use the bathroom. This requires us to take synchronization to a finer granular level rather than implementing it at the method level. So far what we discussed looks like the below when translated into code:

```
0. public class UnisexBathroom {
1.
2.     static String WOMEN = "women";
3.     static String MEN = "men";
4.     static String NONE = "none";
5.
6.     String inUseBy = NONE;
7.     int empsInBathroom = 0;
8.
9.     void useBathroom(String name) throws InterruptedException {
10.         System.out.println(name + " using bathroom. Current employees in bathroom = " + empsInBathroom);
11.         Thread.sleep(10000);
12.         System.out.println(name + " done using bathroom");
13.     }
14.
15.     void maleUseBathroom(String name) throws InterruptedException {
16.
17.         synchronized (this) {
18.             while (inUseBy.equals(WOMEN)) {
19.                 // The wait call will give up the monitor associated
20.                 // with the object, giving other threads a chance to
21.                 // acquire it.
22.                 this.wait();
23.             }
24.             empsInBathroom++;
25.             inUseBy = MEN;
26.         }
27.
28.         useBathroom(name);
29.
30.         synchronized (this) {
```

```

31.         empsInBathroom--;
32.
33.         if (empsInBathroom == 0) inUseBy = NONE;
34.         // Since we might have just updated the value of
35.         // inUseBy, we should notifyAll waiting threads
36.         this.notifyAll();
37.     }
38. }
39.
40. void femaleUseBathroom(String name) throws InterruptedException {
41.
42.     synchronized (this) {
43.         while (inUseBy.equals(MEN)) {
44.             this.wait();
45.         }
46.         empsInBathroom++;
47.         inUseBy = WOMEN;
48.     }
49.
50.     useBathroom(name);
51.
52.     synchronized (this) {
53.         empsInBathroom--;
54.
55.         if (empsInBathroom == 0) inUseBy = NONE;
56.         // Since we might have just updated the value of
57.         // inUseBy, we should notifyAll waiting threads
58.         this.notifyAll();
59.     }
60. }
61.}
```

The code so far allows any number of men or women to gather in the bathroom. However, it allows only one gender to do so. The methods are mirror images of each other with only gender-specific variable changes. Let's discuss the important portions of the code.

- **Lines 17-26:** Since java monitors are mesa monitors, we use a while loop to check for the variable `inUseBy`. If it is set to `MEN` or `NONE` then, we know the bathroom is either empty or already has men and therefore it is safe to proceed ahead. If the `inUseBy` is set to `WOMEN`, then the male thread, invokes `wait()` on **line 23**. Note, the thread would give up the monitor for the object on which it is synchronized thus allowing other threads to synchronize on the same object and maybe update the `inUseBy` variable
- **Line 28** has no synchronization around it. If a male thread reaches here, we are guaranteed that either the bathroom was already in use by men or no one was using it.

- **Lines 30-37:** After using the bathroom, the male thread is about to leave the method so it should remember to decrement the number of occupants in the bathroom. As soon as it does that, it has to check if it were the last member of its gender to leave the bathroom and if so then it should also update the `inUseBy` variable to **NONE**. Finally, the thread notifies any other waiting threads that they are free to check the value of `inUseBy` in case it has updated it. **Question:** Why did we use `notifyAll()` instead of `notify()`?

Now we need to incorporate the logic of limiting the number of employees of a given gender that can be in the bathroom at the same time. Limiting access, intuitively leads one to use a semaphore. A semaphore would do just that - limit access to a fixed number of threads, which in our case is 3.

Complete Code

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

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        UnisexBathroom.runTest();
    }
}

class UnisexBathroom {

    static String WOMEN = "women";
    static String MEN = "men";
    static String NONE = "none";

    String inUseBy = NONE;
    int empsInBathroom = 0;
    Semaphore maxEmps = new Semaphore(3);

    void useBathroom(String name) throws InterruptedException {
        System.out.println("\n" + name + " using bathroom. Current employees in bathroom = " +
                           empsInBathroom + " " + System.currentTimeMillis());
        Thread.sleep(3000);
        System.out.println("\n" + name + " done using bathroom " + System.currentTimeMillis());
    }

    void maleUseBathroom(String name) throws InterruptedException {

        synchronized (this) {
            while (inUseBy.equals(WOMEN)) {
                this.wait();
            }
        }
    }
}
```

```

        maxEmps.acquire();
        empsInBathroom++;
        inUseBy = MEN;
    }

    useBathroom(name);
    maxEmps.release();

    synchronized (this) {
        empsInBathroom--;

        if (empsInBathroom == 0) inUseBy = NONE;
        this.notifyAll();
    }
}

void femaleUseBathroom(String name) throws InterruptedException {

    synchronized (this) {
        while (inUseBy.equals(MEN)) {
            this.wait();
        }
        maxEmps.acquire();
        empsInBathroom++;
        inUseBy = WOMEN;
    }

    useBathroom(name);
    maxEmps.release();

    synchronized (this) {
        empsInBathroom--;

        if (empsInBathroom == 0) inUseBy = NONE;
        this.notifyAll();
    }
}

public static void runTest() throws InterruptedException {

    final UnisexBathroom unisexBathroom = new UnisexBathroom();

    Thread female1 = new Thread(new Runnable() {
        public void run() {
            try {
                unisexBathroom.femaleUseBathroom("Lisa");
            } catch (InterruptedException ie) {

```

```

        }
    }
});

Thread male1 = new Thread(new Runnable() {
    public void run() {
        try {
            unisexBathroom.maleUseBathroom("John");
        } catch (InterruptedException ie) {

        }
    }
});

Thread male2 = new Thread(new Runnable() {
    public void run() {
        try {
            unisexBathroom.maleUseBathroom("Bob");
        } catch (InterruptedException ie) {

        }
    }
});

Thread male3 = new Thread(new Runnable() {
    public void run() {
        try {
            unisexBathroom.maleUseBathroom("Anil");
        } catch (InterruptedException ie) {

        }
    }
});

Thread male4 = new Thread(new Runnable() {
    public void run() {
        try {
            unisexBathroom.maleUseBathroom("Wentao");
        } catch (InterruptedException ie) {

        }
    }
});

female1.start();
male1.start();
male2.start();
male3.start();

```

```

        male4.start();

        female1.join();
        male1.join();
        male2.join();
        male3.join();
        male4.join();

    }
}

```

If you look at the program output, you'd notice that the current employees in the bathroom at one point is printed out to be 4 when the max allowed is 3. This is just an outcome of how the code is structured, read on below for an explanation.

In our test case we have four males and one female aspiring to use the bathroom. We let the female thread use the bathroom first and then let all the male threads loose. From the output, you'll observe, that no male thread is inside the bathroom until Lisa is done using the bathroom. After that, three male threads get access to the bathroom at the same instant. The fourth male thread is held behind, till one of the male thread exits the bathroom.

We acquire the semaphore from within the synchronized block and in case the thread blocks on acquire it doesn't give up the monitor, for the object which implies that `inUseBy` and `empsInBathroom` won't be modified till this blocked thread gets out of the synchronized block. This is a very subtle and important point to understand.

Imagine, if there are already three men in the bathroom and a fourth one comes along then he gets blocked on **line#31**. This thread still holds the bathroom object's monitor, when it becomes dormant due to non-availability of permits. This prevents any female thread from changing the `inUseBy` to **WOMEN** under any circumstance nor can the value of `empsInBathroom` be changed.

Next note the threads returning from the `useBathroom` method, release the semaphore. We must release the semaphore here because if we do not then the blocked fourth male thread would never release the object's monitor and the returning threads will never be able to access the second synchronization block.

On releasing the semaphore, the blocked male thread will increment the `empsInBathroom` variable to 4, before the thread that signaled the semaphore enters the second synchronized block and decrements itself from the count. It is also possible that male threads pile up before the

second synchronized block, while new arriving threads are chosen by the system to run through the first synchronized block. In such a scenario, the count `empsInBathroom` will keep increasing as threads returning from the bathroom wait to synchronize on the `this` object and decrement the count in the second synchronization block. Though eventually, these threads will be able to synchronize and the count will reach zero.

As an alternative, we can put the statement `maxEmps.acquire()` on **line # 35** instead of **line # 31** and the program will continue to work correctly. However, then the variable `empsInBathroom` can potentially take up a value equal to the number of waiting threads. In our current version, the max value the variable `empsInBathroom` can take up is 4.

To prove the correctness of the program, you'll need to convince yourself that the variables involved are all being atomically manipulated.

Also, note that this solution isn't fair to the genders. If the first thread to get bathroom access is male, and before it's done using the bathroom, a steady stream of male threads start to arrive for bathroom use, then any waiting female threads will starve.

Follow up question#

- Try writing a solution in which there's no possibility of starvation for threads of either gender.

Implementing a Barrier

This lesson discusses how a barrier can be implemented in Java.

Problem Statement#

A barrier can be thought of as a point in the program code, which all or some of the threads need to reach at before any one of them is allowed to proceed further.

Working of a Barrier

1. No thread has reached the barrier yet



Size = 3

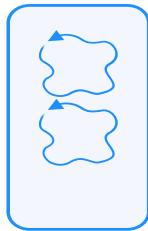
2. The first thread reaching the barrier is blocked



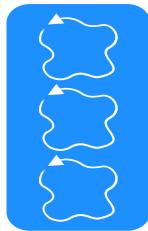
3. A second thread making its way to the barrier



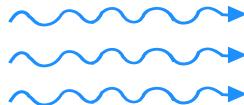
4. Two threads waiting at the barrier for a third one to arrive



5. All threads reach the barrier



6. The barrier releases all threads



Solution#

A barrier allows multiple threads to congregate at a point in code before any one of the threads is allowed to move forward. Java and most other languages provide libraries which make barrier construct available for developer use. Even though we are re-inventing the wheel but this makes for a good interview question.

We can immediately realize that our solution will need a count variable to track the number of threads that have arrived at the barrier. If we have n threads, then $n-1$ threads must wait for the **n^{th}** thread to arrive. This suggests we have the **$n-1$** threads execute the wait method and the **n^{th}** thread wakes up all the asleep **$n-1$** threads.

Below is the code:

```
1. public class Barrier {  
2.  
3.     int count = 0;  
4.     int totalThreads;  
5.  
6.     public Barrier(int totalThreads) {  
7.         this.totalThreads = totalThreads;  
8.     }  
9.  
10.    public synchronized void await() throws InterruptedException {  
11.        // increment the counter whenever a thread arrives at the  
12.        // barrier.  
13.        count++;  
14.  
15.        if (count == totalThreads) {  
16.            // wake up all the threads.  
17.            notifyAll();  
18.            // remember to reset count so that barrier can be reused  
19.            count = 0;  
20.        } else {  
21.            // wait if you aren't the nth thread  
22.            wait();  
23.        }  
24.    }  
25.}
```

Notice how we are resetting the count to zero in **line 19**. This is done so that we are able to re-use the barrier.

Below is the working code, alongwith a test case. The test-case creates three threads and has them synchronize on a barrier three times. We introduce sleeps accordingly so that, thread 1 reaches the barrier first, then thread 2 and finally thread 3. None of the thread is able to move forward until all the threads reach the barrier. This is verified by the order in which each thread prints itself in the output.

First Cut#

Here's the first stab at the problem

```
class Demonstration {  
    public static void main( String args[] ) throws Exception{  
        Barrier.runTest();  
    }  
}  
  
class Barrier {  
  
    int count = 0;  
    int totalThreads;  
  
    public Barrier(int totalThreads) {  
        this.totalThreads = totalThreads;  
    }  
  
    public synchronized void await() throws InterruptedException {  
        count++;  
  
        if (count == totalThreads) {  
            notifyAll();  
            count = 0;  
        } else {  
            wait();  
        }  
    }  
  
    public static void runTest() throws InterruptedException {  
        final Barrier barrier = new Barrier(3);  
  
        Thread p1 = new Thread(new Runnable() {  
            public void run() {  
                try {  
                    System.out.println("Thread 1");  
                    barrier.await();  
                    System.out.println("Thread 1");  
                    barrier.await();  
                    System.out.println("Thread 1");  
                    barrier.await();  
                } catch (InterruptedException ie) {  
                }  
            }  
        });  
  
        Thread p2 = new Thread(new Runnable() {
```

```

public void run() {
    try {
        Thread.sleep(500);
        System.out.println("Thread 2");
        barrier.await();
        Thread.sleep(500);
        System.out.println("Thread 2");
        barrier.await();
        Thread.sleep(500);
        System.out.println("Thread 2");
        barrier.await();
    } catch (InterruptedException ie) {
    }
}
});

Thread p3 = new Thread(new Runnable() {
    public void run() {
        try {
            Thread.sleep(1500);
            System.out.println("Thread 3");
            barrier.await();
            Thread.sleep(1500);
            System.out.println("Thread 3");
            barrier.await();
            Thread.sleep(1500);
            System.out.println("Thread 3");
            barrier.await();
        } catch (InterruptedException ie) {
        }
    }
});
});

p1.start();
p2.start();
p3.start();

p1.join();
p2.join();
p3.join();
}
}

```

When you run the above code, you'll see that the threads print themselves in order i.e. first thread 1 then thread 2 and finally thread 3 prints. Thread 1 after reaching the barrier waits for the other two threads to reach the barrier before moving forward.

The above code has a subtle but very crucial bug! Can you spot the bug and try to fix it before reading on?

Second Cut#

The previous code would have been hunky dory if we were guaranteed that no spurious wake-ups could ever occur. The `wait()` method invocation without the while loop is an error. We discussed in previous sections that `wait()` should always be used with a while loop that checks for a condition and if found false should make the thread wait again.

The condition the while loop can check for is simply how many threads have incremented the `count` variable so far. A thread that wakes up spuriously should go back to sleep if the `count` is less than the total number of threads. We can check for this condition as follows

```
while (count < totalThreads)
    wait();
```

The while loop introduces another problem. When the last thread does a `notifyAll()` it also resets the `count` to 0, which means the threads that are legitimately woken up will always be stuck in the while loop because `count` is immediately set to zero. What we really want is not to reset the `count` variable to zero until all the threads escape the while condition when `count` becomes `totalThreads`. Below is the improved version:

```
1. public class Barrier {
2.     int released = 0;
3.     int count = 0;
4.     int totalThreads;
5.
6.     public Barrier(int totalThreads) {
7.         this.totalThreads = totalThreads;
8.     }
9.
10.    public synchronized void await() throws InterruptedException {
11.        // increment the counter whenever a thread arrives at the
12.        // barrier.
13.        count++;
14.
15.        if (count == totalThreads) {
16.            // wake up all the threads.
17.            notifyAll();
18.            // remember to reset count so that barrier can be reused
19.            released = totalThreads;
20.        } else {
21.            // wait till all threads reach barrier
22.            while (count < totalThreads)
23.                wait();
24.        }
25.    }
}
```

```
26.     released--;
27.     if (released == 0) count = 0;
28. }
29.}
```

The above code introduces a new variable `released` that keeps tracks of how many threads exit the barrier and when the last thread exits the barrier it resets `count` to zero, so that the barrier object can be reused in the future.

There is still a bug in the above code! Can you guess what it is?

Final Cut#

To understand why the above code is broken, consider three threads **t1**, **t2**, and **t3** trying to `await()` on a barrier object in an infinite loop. Note the following sequence of events

1. Threads t1 and t2 invoke `await()` and end up waiting at **line#23**. The count variable is set to 2 and any spurious wakeups will cause t1 and t2 to go back to waiting.
2. Threads t3 comes along, executes the if block on **line#15** and finds `count == totalThreads`. Thread t3 doesn't wait, notifies threads t1 and t2 to wakeup and exits.
3. **If thread t3 attempts to invoke `await()` immediately after exiting it and is also granted the monitor before threads t1 or t2 get a chance to acquire the monitor then the count variable will be incremented to 4.**
4. With `count` equal to 4, t3 will not block at the barrier and exit which breaks the contract for the barrier.
5. The invocation order of the `await()` method was t1,t2,t3, and t3 again. The right behavior would have been to release t1,t2, or t3 in any order and then block t3 on its second invocation of the `await()` method.
6. Another flaw with the above code is, it can cause a deadlock. Suppose we wanted the three threads t1, t2, and t3 to congregate at a barrier twice. The first invocation was in the order [t1, t2, t3] and the second was in the order [t3, t2, t1]. If t3 immediately invoked await after the first barrier, it would go past the second barrier without stopping while t2 and t1 would become stranded at the second barrier, since `count` would never equal `totalThreads`.

The fix requires us to block any new threads from proceeding until all the threads that have reached the previous barrier are released. The code with the fix appears below:

```
1. public class Barrier {  
2.     int released = 0;  
3.     int count = 0;  
4.     int totalThreads;  
5.  
6.     public Barrier(int totalThreads) {  
7.         this.totalThreads = totalThreads;  
8.     }  
9.  
10.    public synchronized void await() throws InterruptedException {  
11.  
12.        // block any new threads from proceeding till,  
13.        // all threads from previous barrier are released  
14.        while (count == totalThreads) wait();  
15.  
16.        // increment the counter whenever a thread arrives at the  
17.        // barrier.  
18.        count++;  
19.  
20.        if (count == totalThreads) {  
21.            // wake up all the threads.  
22.            notifyAll();  
23.            // remember to set released to totalThreads  
24.            released = totalThreads;  
25.        } else {  
26.            // wait till all threads reach barrier  
27.            while (count < totalThreads)  
28.                wait();  
29.        }  
30.  
31.        released--;  
32.        if (released == 0) {  
33.            count = 0;  
34.            // remember to wakeup any threads  
35.            // waiting on line#14  
36.            notifyAll();  
37.        }  
38.    }  
39.}
```

Complete code:

```
class Demonstration {  
    public static void main( String args[] ) throws Exception{  
        Barrier.runTest();  
    }  
}  
  
class Barrier {
```

```

int count = 0;
int released = 0;
int totalThreads;

public Barrier(int totalThreads) {
    this.totalThreads = totalThreads;
}

public static void runTest() throws InterruptedException {
    final Barrier barrier = new Barrier(3);

    Thread p1 = new Thread(new Runnable() {
        public void run() {
            try {
                System.out.println("Thread 1");
                barrier.await();
                System.out.println("Thread 1");
                barrier.await();
                System.out.println("Thread 1");
                barrier.await();
            } catch (InterruptedException ie) {
            }
        }
    });
};

Thread p2 = new Thread(new Runnable() {
    public void run() {
        try {
            Thread.sleep(500);
            System.out.println("Thread 2");
            barrier.await();
            Thread.sleep(500);
            System.out.println("Thread 2");
            barrier.await();
            Thread.sleep(500);
            System.out.println("Thread 2");
            barrier.await();
        } catch (InterruptedException ie) {
        }
    }
});
};

Thread p3 = new Thread(new Runnable() {
    public void run() {
        try {
            Thread.sleep(1500);
            System.out.println("Thread 3");
        }
    }
});

```

```
        barrier.await();
        Thread.sleep(1500);
        System.out.println("Thread 3");
        barrier.await();
        Thread.sleep(1500);
        System.out.println("Thread 3");
        barrier.await();
    } catch (InterruptedException ie) {
    }
}
});

p1.start();
p2.start();
p3.start();

p1.join();
p2.join();
p3.join();
}

public synchronized void await() throws InterruptedException {

    while (count == totalThreads)
        wait();

    count++;

    if (count == totalThreads) {
        notifyAll();
        released = totalThreads;
    } else {

        while (count < totalThreads)
            wait();
    }

    released--;
    if (released == 0) {
        count = 0;
        // remember to wakeup any threads
        // waiting on line#81
        notifyAll();
    }
}
}
```

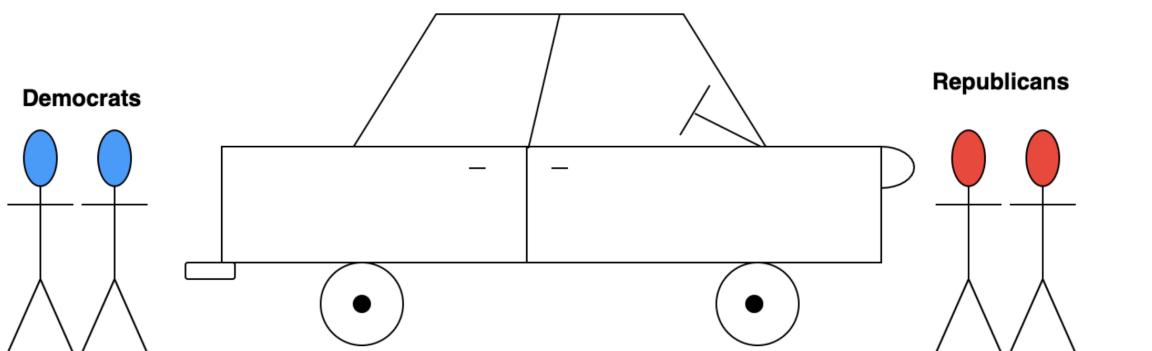
Uber Ride Problem

This lesson solves the constraints of an imaginary Uber ride problem where Republicans and Democrats can't be seated as a minority in a four passenger car.

Problem Statement#

Imagine at the end of a political conference, republicans and democrats are trying to leave the venue and ordering Uber rides at the same time. However, to make sure no fight breaks out in an Uber ride, the software developers at Uber come up with an algorithm whereby either an Uber ride can have all democrats or republicans or two Democrats and two Republicans. All other combinations can result in a fist-fight.

Your task as the Uber developer is to model the ride requestors as threads. Once an acceptable combination of riders is possible, threads are allowed to proceed to ride. Each thread invokes the method `seated()` when selected by the system for the next ride. When all the threads are seated, any one of the four threads can invoke the method `drive()` to inform the driver to start the ride.



Solution#

First let us model the problem as a class. We'll have two methods one called by a Democrat and one by a Republican to get a ride home. When either one gets a seat on the next ride, it'll call the `seated()` method.

To make up an allowed combination of riders, we'll need to keep a count of Democrats and Republicans who have requested for rides. We create two variables for this purpose and modify them within a lock/mutex. In this problem, we'll use the `ReentrantLock` class provided by java's `util.concurrent` package when manipulating counts for democrats and republicans.

Realize we'll also need a barrier where all the four threads, that have been selected for the Uber ride arrive at, before riding away. This is analogous to the four riders being seated in the car and the doors being shut.

Once the doors are shut, one of the riders has to tell the driver to drive which we simulate with a call to the `drive()` method. Note that exactly one thread makes the shout-out to the driver to `drive()`.

The initial class skeleton looks like the following:

```
public class UberSeatingProblem {

    private int republicans = 0;
    private int democrats = 0;

    CyclicBarrier barrier = new CyclicBarrier(4);
    ReentrantLock lock = new ReentrantLock();

    void seatDemocrat() throws InterruptedException, BrokenBarrierException {
    }

    void seatRepublican() throws InterruptedException, BrokenBarrierException {
    }

    void seated() {
    }

    void drive() {
}
}
```

Let's focus on the `seatDemocrat()` method first. For simplicity imagine the first thread is a democrat and invokes `seatDemocrat()`. Since there's no other rider available, it should be put to wait. We can use a semaphore to make this thread wait. We'll not use a barrier, because we don't know what party loyalty the threads arriving in future would have. It might be that the next four threads are all republican and this Democrat isn't placed on the next Uber ride. To differentiate between waiting democrats and waiting republicans, we'll use two different semaphores `demsWaiting` and `repubsWaiting`. Our first democrat thread will end up `acquire()`-ing the `demsWaiting` semaphore.

Now it's easy to reason about how we select the threads for a ride. A democrat thread has to check the following cases:

- If there are already 3 waiting democrats, then we signal the `demsWaiting` three times so that all these four democrats can ride together in the next Uber ride.

- If there are two or more republican threads waiting and at least two democrat threads (including the current thread) waiting, then the current democrat thread can signal the `repubsWaiting` semaphore twice to release the two waiting republican threads and signal the `demsWaiting` semaphore once to release one more democrat thread. Together the four of them would make up the next ride consisting of two republican and two democrats.
- If the above two conditions aren't true then the current democrat thread should simply wait itself at the `demsWaiting` semaphore and release the lock object so that other threads can enter the critical sections.

The logic we discussed so far is translated into code below:

```

void seatDemocrat() throws InterruptedException, BrokenBarrierException {
    boolean rideLeader = false;
    lock.lock();

    democrats++;

    if (democrats == 4) {
        // Seat all the democrats in the Uber ride.
        demsWaiting.release(3);
        democrats -= 4;
        rideLeader = true;
    } else if (democrats == 2 && republicans >= 2) {
        // Seat 2 democrats & 2 republicans
        demsWaiting.release(1);
        repubsWaiting.release(2);
        rideLeader = true;
        democrats -= 2;
        republicans -= 2;
    } else {
        lock.unlock();
        demsWaiting.acquire();
    }

    seated();
    barrier.await();

    if (rideLeader == true) {
        drive();
        lock.unlock();
    }
}

```

The thread that signals other threads to come along for the ride marks itself as the `rideLeader`. This thread is responsible for informing the driver to `drive()`. We can come up with some other criteria to choose the rider

leader but given the logic we implemented, it is easiest to make the thread that determines an acceptable ride combination as the ride leader.

The republicans' `seatRepublican()` method is analogous to the `seatDemocrat()` method.

Complete Code#

The complete code appears below:

```
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.ReentrantLock;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        UberSeatingProblem.runTest();
    }
}

class UberSeatingProblem {

    private int republicans = 0;
    private int democrats = 0;

    private Semaphore demsWaiting = new Semaphore(0);
    private Semaphore repubsWaiting = new Semaphore(0);

    CyclicBarrier barrier = new CyclicBarrier(4);
    ReentrantLock lock = new ReentrantLock();

    void drive() {
        System.out.println("Uber Ride on Its wayyyy... with ride leader " +
Thread.currentThread().getName());
        System.out.flush();
    }

    void seatDemocrat() throws InterruptedException, BrokenBarrierException {

        boolean rideLeader = false;
        lock.lock();

        democrats++;

        if (democrats == 4) {
```

```

// Seat all the democrats in the Uber ride.
demsWaiting.release(3);
democrats -= 4;
rideLeader = true;
} else if (democrats == 2 && republicans >= 2) {
    // Seat 2 democrats & 2 republicans
    demsWaiting.release(1);
    repubsWaiting.release(2);
    rideLeader = true;
    democrats -= 2;
    republicans -= 2;
} else {
    lock.unlock();
    demsWaiting.acquire();
}
}

seated();
barrier.await();

if (rideLeader == true) {
    drive();
    lock.unlock();
}
}

void seated() {
    System.out.println(Thread.currentThread().getName() + " seated");
    System.out.flush();
}

void seatRepublican() throws InterruptedException, BrokenBarrierException {

    boolean rideLeader = false;
    lock.lock();

    republicans++;

    if (republicans == 4) {
        // Seat all the republicans in the Uber ride.
        repubsWaiting.release(3);
        rideLeader = true;
        republicans -= 4;
    } else if (republicans == 2 && democrats >= 2) {
        // Seat 2 democrats & 2 republicans
        repubsWaiting.release(1);
        demsWaiting.release(2);
        rideLeader = true;
        republicans -= 2;
    }
}

```

```

democrats -= 2;
} else {
    lock.unlock();
    repubsWaiting.acquire();
}

seated();
barrier.await();

if (rideLeader) {
    drive();
    lock.unlock();
}
}

public static void runTest() throws InterruptedException {

final UberSeatingProblem uberSeatingProblem = new UberSeatingProblem();
Set<Thread> allThreads = new HashSet<Thread>();

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

    Thread thread = new Thread(new Runnable() {
        public void run() {
            try {
                uberSeatingProblem.seatDemocrat();
            } catch (InterruptedException ie) {
                System.out.println("We have a problem");

            } catch (BrokenBarrierException bbe) {
                System.out.println("We have a problem");
            }
        }
    });
    thread.setName("Democrat_" + (i + 1));
    allThreads.add(thread);

    Thread.sleep(50);
}

for (int i = 0; i < 14; i++) {
    Thread thread = new Thread(new Runnable() {
        public void run() {
            try {
                uberSeatingProblem.seatRepublican();
            } catch (InterruptedException ie) {

```

```
System.out.println("We have a problem");

} catch (BrokenBarrierException bbe) {
    System.out.println("We have a problem");
}

}

};

thread.setName("Republican_" + (i + 1));
allThreads.add(thread);
Thread.sleep(20);
}

for (Thread t : allThreads) {
    t.start();
}

for (Thread t : allThreads) {
    t.join();
}

}
```

The output of the program will show the members of each ride. Since we create four more republican threads than democrat threads, you should see at least one ride with all republican riders.

The astute reader may wonder what factor determines that a ride is evenly split between members of the two parties or entirely made up of the members of the same party, given enough riders exist that both combinations can be possible. The key is to realize that each thread enters the critical sections `seatDemocrat()` or `seatRepublican()` one at a time because of the lock at the beginning of the two methods. Whether a ride is evenly split between the two types of riders or consists entirely of one type of riders depends upon the order in which the threads enter the critical section. For instance, if we create four democrat and four republican threads then we can either get two rides each split between the two types or two rides each made of the same type. If the first four threads to sequentially enter the critical sections are democrat, then the two rides will be made up of either entirely democrats or republicans. Since threads are scheduled for execution non-deterministically, we can't be certain what would be the makeup of each ride.

Dining Philosophers

This chapter discusses the famous Dijkstra's Dining Philosopher's problem. Two different solutions are explained at length.

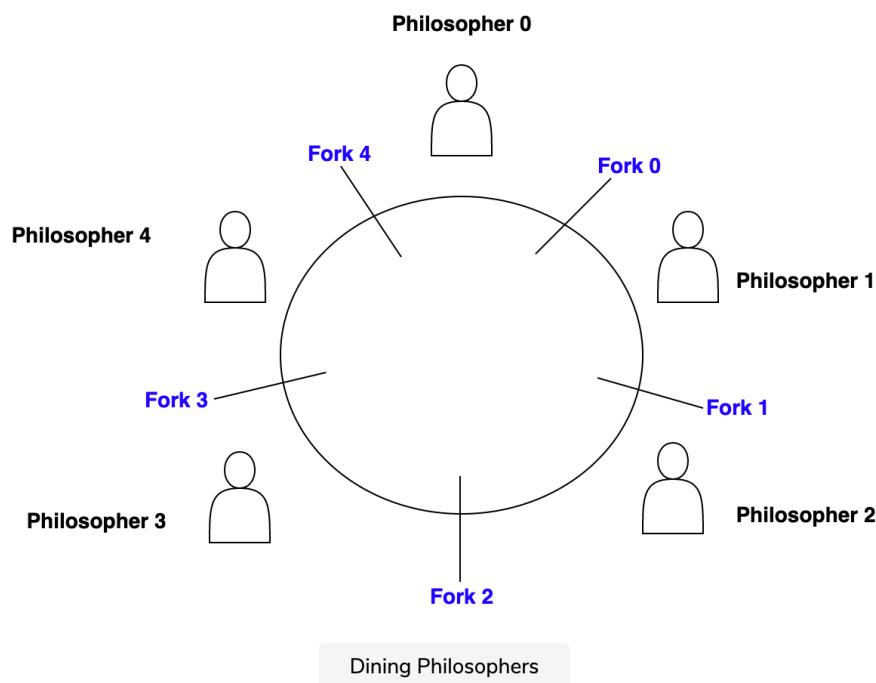
Problem Statement#

This is a classical synchronization problem proposed by Dijkstra.

Imagine you have five philosopher's sitting on a roundtable. The philosopher's do only two kinds of activities. One they contemplate, and two they eat. However, they have only five forks between themselves to eat their food with. Each philosopher requires both the fork to his left and the fork to his right to eat his food.

The arrangement of the philosophers and the forks are shown in the diagram.

Design a solution where each philosopher gets a chance to eat his food without causing a deadlock



Solution#

For no deadlock to occur at all and have all the philosopher be able to eat, we would need ten forks, two for each philosopher. With five forks available, at most, only two philosophers will be able to eat while letting a third hungry philosopher to hold onto the fifth fork and wait for another one to become available before he can eat.

Think of each fork as a resource that needs to be owned by one of the philosophers sitting on either side.

Let's try to model the problem in code before we even attempt to find a solution. Each fork represents a resource that two of the philosophers on either side can attempt to acquire. This intuitively suggests using a semaphore with a permit value of 1 to represent a fork. Each philosopher can then be thought of as a thread that tries to acquire the forks to the left and right of it. Given this, let's see how our class would look like.

```
public class DiningPhilosophers {  
  
    // This random variable is used for test purposes only  
    private static Random random = new Random(System.currentTimeMillis());  
    // Five semaphore represent the five forks.  
    private Semaphore[] forks = new Semaphore[5];  
  
    // Initializing the semaphores with a permit of 1  
    public DiningPhilosophers() {  
        forks[0] = new Semaphore(1);  
        forks[1] = new Semaphore(1);  
        forks[2] = new Semaphore(1);  
        forks[3] = new Semaphore(1);  
        forks[4] = new Semaphore(1);  
    }  
  
    // Represents how a philosopher lives his life  
    public void lifecycleOfPhilosopher(int id) throws InterruptedException {  
        while (true) {  
            contemplate();  
            eat(id);  
        }  
    }  
  
    // We can sleep the thread when the philosopher is thinking  
    void contemplate() throws InterruptedException {  
        Thread.sleep(random.nextInt(500));  
    }  
  
    // This method will have the meat of the solution, where the  
    // philosopher is trying to eat.  
    void eat(int id) throws InterruptedException {  
    }  
}
```

That was easy enough. Now think about the eat method, when a philosopher wants to eat, he needs the fork to the left and right of him. So:

- Philosopher A(0) needs forks 4 and 0
- Philosopher B(1) needs forks 0 and 1
- Philosopher C(2) needs forks 1 and 2
- Philosopher D(3) needs forks 2 and 3
- Philosopher E(4) needs forks 3 and 4

This means each thread (philosopher) will also need to tell us what ID it is before we can attempt to lock the appropriate forks for him. That is why you see the `eat()` method take in an ID parameter.

We can programmatically express the requirement for each philosopher to hold the right and left forks as follows:

```
forks[id]
forks[(id+4) % 5]
```

So far we haven't discussed deadlocks and without them the naive solution would look like the following:

```
public class DiningPhilosophers {

    private static Random random = new Random(System.currentTimeMillis());

    private Semaphore[] forks = new Semaphore[5];
    private Semaphore maxDiners = new Semaphore(4);

    public DiningPhilosophers() {
        forks[0] = new Semaphore(1);
        forks[1] = new Semaphore(1);
        forks[2] = new Semaphore(1);
        forks[3] = new Semaphore(1);
        forks[4] = new Semaphore(1);
    }

    public void lifecycleOfPhilosopher(int id) throws InterruptedException {
        while (true) {
            contemplate();
            eat(id);
        }
    }

    void contemplate() throws InterruptedException {
        Thread.sleep(random.nextInt(500));
    }

    void eat(int id) throws InterruptedException {
        // acquire the left fork first
        forks[id].acquire();

        // acquire the right fork second
        forks[(id + 4) % 5].acquire();

        // eat to your heart's content
        System.out.println("Philosopher " + id + " is eating");

        // release forks for others to use
        forks[id].release();
        forks[(id + 4) % 5].release();
    }
}
```

```
    }
```

If you run the above code eventually, it'll at some point end up in a deadlock. Realize if all the philosophers simultaneously grab their left fork, none would be able to eat. Below we discuss a couple of ways to avoid this deadlock and arrive at the final solution.

Limiting philosophers about to eat#

A very simple fix is to allow only four philosophers at any given point in time to even try to acquire forks. Convince yourself that with five forks and four philosophers deadlock is impossible, since at any point in time, even if each philosopher grabs one fork, there will still be one fork left that can be acquired by one of the philosophers to eat. Implementing this solution requires us to introduce another semaphore with a permit of 4 which guards the logic for lifting/grabbing of the forks by the philosophers. The code appears below.

```
public class DiningPhilosophers {

    private static Random random = new Random(System.currentTimeMillis());

    private Semaphore[] forks = new Semaphore[5];
    private Semaphore maxDiners = new Semaphore(4);

    public DiningPhilosophers() {
        forks[0] = new Semaphore(1);
        forks[1] = new Semaphore(1);
        forks[2] = new Semaphore(1);
        forks[3] = new Semaphore(1);
        forks[4] = new Semaphore(1);
    }

    public void lifecycleOfPhilosopher(int id) throws InterruptedException {
        while (true) {
            contemplate();
            eat(id);
        }
    }

    void contemplate() throws InterruptedException {
        Thread.sleep(random.nextInt(500));
    }

    void eat(int id) throws InterruptedException {
        // maxDiners allows only 4 philosophers to
        // attempt picking up forks.
        maxDiners.acquire();
    }
}
```

```

        forks[id].acquire();
        forks[(id + 1) % 5].acquire();
        System.out.println("Philosopher " + id + " is eating");
        forks[id].release();
        forks[(id + 1) % 5].release();

        maxDiners.release();
    }
}

```

Ordering of fork pick-up#

Another solution is to make any one of the philosophers pick-up the left fork first instead of the right one. If this gentleman successfully acquires the left fork then, it implies:

- The philosopher sitting next to the left-handed gentleman can't acquire his right fork, so he's blocked from eating since he must first pick up the fork to the right of him (already held by the left-handed philosopher). The blocked philosopher's left fork is free to be picked up by another philosopher.
- The left-handed philosopher may acquire his right fork implying no deadlock since he already picked up his left fork first. Or if he's unable to acquire his right fork, then the gentleman previous to the left-handed philosopher in an anti-clockwise direction will necessarily have had acquired both his right and left forks and will eat. Again, not resulting in a deadlock.

It doesn't matter which philosopher is chosen to be left-handed and made to pick up his left fork first instead of the right one since its a circle. In our solution, we select the philosopher with id=3 as the left-handed philosopher

```

public class DiningPhilosophers2 {

    private static Random random = new Random(System.currentTimeMillis());

    private Semaphore[] forks = new Semaphore[5];

    public DiningPhilosophers2() {
        forks[0] = new Semaphore(1);
        forks[1] = new Semaphore(1);
        forks[2] = new Semaphore(1);
        forks[3] = new Semaphore(1);
        forks[4] = new Semaphore(1);
    }

    public void lifecycleOfPhilosopher(int id) throws InterruptedException
    {

```

```

        while (true) {
            contemplate();
            eat(id);
        }
    }

    void contemplate() throws InterruptedException {
        Thread.sleep(random.nextInt(500));
    }

    void eat(int id) throws InterruptedException {

        // We randomly selected the philosopher with
        // id 3 as left-handed. All others must be
        // right-handed to avoid a deadlock.
        if (id == 3) {
            acquireForkLeftHanded(3);
        } else {
            acquireForkForRightHanded(id);
        }

        System.out.println("Philosopher " + id + " is eating");
        forks[id].release();
        forks[(id + 1) % 5].release();
    }

    void acquireForkForRightHanded(int id) throws InterruptedException {
        forks[id].acquire();
        forks[(id + 1) % 5].acquire();
    }

    // Left-handed philosopher picks the left fork first and then
    // the right one.
    void acquireForkLeftHanded(int id) throws InterruptedException {
        forks[(id + 1) % 5].acquire();
        forks[id].acquire();
    }
}

```

Complete Code 1st Solution#

Below is the code for the first solution we discussed, along with a test. The philosopher threads are perpetual so the widget execution times out. For the limited time the test runs, one can see all philosopher's take turns to eat food without any deadlock.

```

import java.util.Random;
import java.util.concurrent.Semaphore;

class Demonstration {

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

```

```
}

class DiningPhilosophers {

    private static Random random = new Random(System.currentTimeMillis());

    private Semaphore[] forks = new Semaphore[5];
    private Semaphore maxDiners = new Semaphore(4);

    public DiningPhilosophers() {
        forks[0] = new Semaphore(1);
        forks[1] = new Semaphore(1);
        forks[2] = new Semaphore(1);
        forks[3] = new Semaphore(1);
        forks[4] = new Semaphore(1);
    }

    public void lifecycleOfPhilosopher(int id) throws InterruptedException {

        while (true) {
            contemplate();
            eat(id);
        }
    }

    void contemplate() throws InterruptedException {
        Thread.sleep(random.nextInt(50));
    }

    void eat(int id) throws InterruptedException {
        maxDiners.acquire();

        forks[id].acquire();
        forks[(id + 1) % 5].acquire();
        System.out.println("Philosopher " + id + " is eating");
        forks[id].release();
        forks[(id + 1) % 5].release();

        maxDiners.release();
    }

    static void startPhilosopher(DiningPhilosophers dp, int id) {
        try {
            dp.lifecycleOfPhilosopher(id);
        } catch (InterruptedException ie) {

        }
    }
}
```

```
public static void runTest() throws InterruptedException {
    final DiningPhilosophers dp = new DiningPhilosophers();

    Thread p1 = new Thread(new Runnable() {

        public void run() {
            startPhilosopher(dp, 0);
        }
    });

    Thread p2 = new Thread(new Runnable() {

        public void run() {
            startPhilosopher(dp, 1);
        }
    });

    Thread p3 = new Thread(new Runnable() {

        public void run() {
            startPhilosopher(dp, 2);
        }
    });

    Thread p4 = new Thread(new Runnable() {

        public void run() {
            startPhilosopher(dp, 3);
        }
    });

    Thread p5 = new Thread(new Runnable() {

        public void run() {
            startPhilosopher(dp, 4);
        }
    });

    p1.start();
    p2.start();
    p3.start();
    p4.start();
    p5.start();

    p1.join();
    p2.join();
    p3.join();
```

```
    p4.join();
    p5.join();
}
}
```

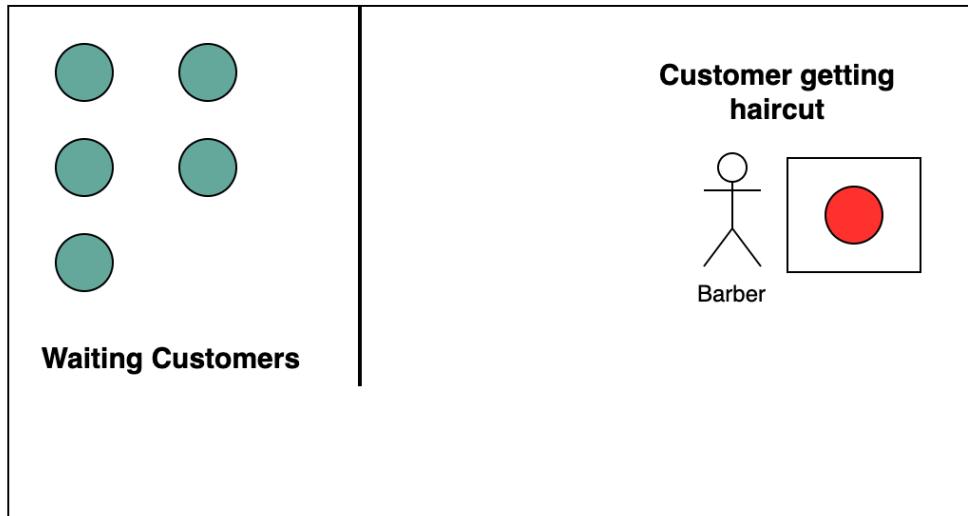
Barber Shop

This lesson visits the synchronization issues when programmatically modeling a hypothetical barber shop and how they are solved using Java's concurrency primitives.

Problem Statement#

A similar problem appears in Silberschatz and Galvin's OS book, and variations of this problem exist in the wild.

A barbershop consists of a waiting room with n chairs, and a barber chair for giving haircuts. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the interaction between the barber and the customers.



Barber Shop Problem

Solution#

First of all, we need to understand the different state transitions for this problem before we devise a solution. Let's look at them piecemeal:

- A customer enters the shop and if all N chairs are occupied, he leaves. This hints at maintaining a count of the waiting customers.
- If any of the N chairs is free, the customer takes up the chair to wait for his turn. Note this translates to using a semaphore on which threads that have found a free chair wait on before being called in by the barber for a haircut.
- If a customer enters the shop and the barber is asleep it implies there are no customers in the shop. The just-entered customer thread wakes up the barber thread. This sounds like using a signaling construct to wake up the barber thread.

We'll have a class which will expose two APIs one for the barber thread to execute and the other for customers. The skeleton of the class would look like the following:

```
public class BarberShopProblem {

    final int CHAIRS = 3;
    int waitingCustomers = 0;
    ReentrantLock lock = new ReentrantLock();

    void customerWalksIn() throws InterruptedException {
    }

    void barber() throws InterruptedException {
    }
}
```

Now let's think about the customer thread. It enters the shop, acquires a lock to test the value of the counter `waitingCustomers`. We must test the value of this variable while no other thread can modify its value, hinting that we'll wrap the test under a lock. If the value equals all the chairs available, then the customer thread gives up the lock and returns from the method. If a chair is available the customer thread increments the variable `waitingCustomers`. Remember, the barber might be asleep which can be modeled as the barber thread waiting on a semaphore `waitForCustomerToEnter`. The customer thread must signal the semaphore `waitForCustomerToEnter` in case the barber is asleep.

Next, the customer thread itself needs to wait on a semaphore before the barber comes over, greets the customer and leads him to the salon chair. Let's call this semaphore `waitForBarberToGetReady`. This is the same semaphore the barber signals as soon as it wakes up. All customer threads waiting for a haircut will block on this `waitForBarberToGetReady` semaphore. The barber signaling this semaphore is akin to letting one customer come

through and sit on the barber chair for a haircut. This logic when coded looks like the following:

```
void customerWalksIn() throws InterruptedException {

    lock.lock();
    if (waitingCustomers == CHAIRS) {
        System.out.println("Customer walks out, all chairs occupied.")
    ;
        // Remember to unlock before leaving
        lock.unlock();
        return;
    }
    waitingCustomers++;
    lock.unlock();

    // Let the barber know you are here, in case he's asleep
    waitForCustomerToEnter.release();
    // Wait for the barber to come take you to the salon chair when it
    s your turn
    waitForBarberToGetReady.acquire();
    // TODO: complete the rest of the logic.
}
```

Now let's work with the barber code. This should be a perpetual loop, where the barber initially waits on the semaphore `waitForCustomerToEnter` to simulate no customers in the shop. If woken up, then it implies that there's at least one customer in the shop who needs a hair-cut and the barber gets up, greets the customer and leads him to his chair before starting the haircut. This sequence is translated into code as the barber thread signaling the `waitForBarberToGetReady` semaphore. Next, the barber simulates a haircut by sleeping for 50 milliseconds

Once the haircut is done. The barber needs to inform the customer thread too; it does so by signaling the `waitForBarberToCutHair` semaphore. The customer thread should already be waiting on this semaphore.

Finally, to make the barber thread know that the current customer thread has left the barber chair and the barber can bring in the next customer, we make the barber thread wait on yet another semaphore `waitForCustomerToLeave`. This is the same semaphore the customer thread needs to signal before exiting. The barber thread's implementation appears below:

```
void barber() throws InterruptedException {

    while (true) {
        // wait till a customer enters a shop
        waitForCustomerToEnter.acquire();
        // let the customer know barber is ready
        waitForBarberToGetReady.release();
    }
}
```

```

        System.out.println("Barber cutting hair...");
        Thread.sleep(50);

        // let customer thread know, haircut is done
        waitForBarberToCutHair.release();
        // wait for customer to leave the barber chair
        waitForCustomerToLeave.acquire();
    }
}

```

The complete customer thread code appears below:

```

void customerWalksIn() throws InterruptedException {

    lock.lock();
    if (waitingCustomers == CHAIRS) {
        System.out.println("Customer walks out, all chairs occupied");
        lock.unlock();
        return;
    }
    waitingCustomers++;
    lock.unlock();

    // Let the barber know, there's atleast 1 customer
    waitForCustomerToEnter.release();
    // Wait for barber to greet you and lead you to barber chair
    waitForBarberToGetReady.acquire();

    // This is where the customer gets the haircut

    // Wait for haircut to complete
    waitForBarberToCutHair.acquire();
    // Leave the barber chair and let barber thread know chair is vaca
nt
    waitForCustomerToLeave.release();

    lock.lock();
    waitingCustomers--;
    lock.unlock();
}

```

Complete Code#

The entire code alongwith the test appears below. Since the barber thread is perpetual, the widget execution would time-out.

```

import java.util.HashSet;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.ReentrantLock;

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

```

```
    BarberShopProblem.runTest();
}
}

class BarberShopProblem {

    final int CHAIRS = 3;
    Semaphore waitForCustomerToEnter = new Semaphore(0);
    Semaphore waitForBarberToGetReady = new Semaphore(0);
    Semaphore waitForCustomerToLeave = new Semaphore(0);
    Semaphore waitForBarberToCutHair = new Semaphore(0);
    int waitingCustomers = 0;
    ReentrantLock lock = new ReentrantLock();
    int hairCutsGiven = 0;

    void customerWalksIn() throws InterruptedException {

        lock.lock();
        if (waitingCustomers == CHAIRS) {
            System.out.println("Customer walks out, all chairs occupied");
            lock.unlock();
            return;
        }
        waitingCustomers++;
        lock.unlock();

        waitForCustomerToEnter.release();
        waitForBarberToGetReady.acquire();

        waitForBarberToCutHair.acquire();
        waitForCustomerToLeave.release();

        lock.lock();
        waitingCustomers--;
        lock.unlock();
    }

    void barber() throws InterruptedException {

        while (true) {
            waitForCustomerToEnter.acquire();
            waitForBarberToGetReady.release();
            hairCutsGiven++;
            System.out.println("Barber cutting hair..." + hairCutsGiven);
            Thread.sleep(50);
            waitForBarberToCutHair.release();
            waitForCustomerToLeave.acquire();
        }
    }
}
```

```
}

public static void runTest() throws InterruptedException {

    HashSet<Thread> set = new HashSet<Thread>();
    final BarberShopProblem barberShopProblem = new BarberShopProblem();

    Thread barberThread = new Thread(new Runnable() {
        public void run() {
            try {
                barberShopProblem.barber();
            } catch (InterruptedException ie) {

            }
        }
    });
    barberThread.start();

    for (int i = 0; i < 10; i++) {
        Thread t = new Thread(new Runnable() {
            public void run() {
                try {
                    barberShopProblem.customerWalksIn();
                } catch (InterruptedException ie) {

                }
            }
        });
        set.add(t);
    }

    for (Thread t : set) {
        t.start();
    }

    for (Thread t : set) {
        t.join();
    }

    set.clear();
    Thread.sleep(800);

    for (int i = 0; i < 5; i++) {
        Thread t = new Thread(new Runnable() {
            public void run() {
                try {
                    barberShopProblem.customerWalksIn();
                } catch (InterruptedException ie) {

                }
            }
        });
    }
}
```

```

        }
    }
});
set.add(t);
}
for (Thread t : set) {
    t.start();
}

barberThread.join();
}
}

```

The execution output would show 6 customers getting a haircut and the rest walking out since there are only three chairs available at the barber shop.

Note we only decrement `waitingCustomers` after the customer thread has received a haircut. However, you may argue that since the customer getting the haircut has left one spot open in the waiting area, it should be possible to have one more thread come in the shop and wait for a total of four threads. Three threads wait on chairs in the waiting area and one thread occupies the barber chair where it undergoes a hair-cut. If we tweak our implementation to compensate for this change (**lines 37, 38, 39** in the below widget) then we'll see the above test give eight haircuts instead of six. The change entails we decrement the `waitingCustomers` variable right after the barber seats a customer. The code with the change appears below. If you run the widget, you'll see eight threads getting haircut.

```

import java.util.HashSet;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.ReentrantLock;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        BarberShopProblem.runTest();
    }
}

class BarberShopProblem {

    final int CHAIRS = 3;
    Semaphore waitForCustomerToEnter = new Semaphore(0);
    Semaphore waitForBarberToGetReady = new Semaphore(0);
    Semaphore waitForCustomerToLeave = new Semaphore(0);
    Semaphore waitForBarberToCutHair = new Semaphore(0);
}

```

```

int waitingCustomers = 0;
ReentrantLock lock = new ReentrantLock();
int hairCutsGiven = 0;

void customerWalksIn() throws InterruptedException {

    lock.lock();
    if (waitingCustomers == CHAIRS) {
        System.out.println("Customer walks out, all chairs occupied");
        lock.unlock();
        return;
    }
    waitingCustomers++;
    lock.unlock();

    waitForCustomerToEnter.release();
    waitForBarberToGetReady.acquire();

    // The chair in the waiting area becomes available
    lock.lock();
    waitingCustomers--;
    lock.unlock();

    waitForBarberToCutHair.acquire();
    waitForCustomerToLeave.release();
}

void barber() throws InterruptedException {

    while (true) {
        waitForCustomerToEnter.acquire();
        waitForBarberToGetReady.release();
        hairCutsGiven++;
        System.out.println("Barber cutting hair..." + hairCutsGiven);
        Thread.sleep(50);
        waitForBarberToCutHair.release();
        waitForCustomerToLeave.acquire();
    }
}

public static void runTest() throws InterruptedException {

    HashSet<Thread> set = new HashSet<Thread>();
    final BarberShopProblem barberShopProblem = new BarberShopProblem();

    Thread barberThread = new Thread(new Runnable() {
        public void run() {
            try {

```

```

        barberShopProblem.barber();
    } catch (InterruptedException ie) {

    }
}
});
barberThread.start();

for (int i = 0; i < 10; i++) {
    Thread t = new Thread(new Runnable() {
        public void run() {
            try {
                barberShopProblem.customerWalksIn();
            } catch (InterruptedException ie) {

            }
        }
    });
    set.add(t);
}

for (Thread t : set) {
    t.start();
}

for (Thread t : set) {
    t.join();
}

set.clear();
Thread.sleep(500);

for (int i = 0; i < 5; i++) {
    Thread t = new Thread(new Runnable() {
        public void run() {
            try {
                barberShopProblem.customerWalksIn();
            } catch (InterruptedException ie) {

            }
        }
    });
    set.add(t);
}
for (Thread t : set) {
    t.start();
    Thread.sleep(5);
}

```

```
    barberThread.join();
}
}
```

Superman Problem

This lesson is about correctly implementing a singleton pattern in Java

Problem Statement#

You are designing a library of superheroes for a video game that your fellow developers will consume. Your library should always create a single instance of any of the superheroes and return the same instance to all the requesting consumers.

Say, you start with the class `Superman`. Your task is to make sure that other developers using your class can never instantiate multiple copies of superman. After all, there is only one superman!

Solution#

You probably guessed we are going to use the **singleton** pattern to solve this problem. The singleton pattern sounds very naive and simple but when it comes to implementing it correctly in Java, it's no cakewalk.

First let us understand what the pattern is. A singleton pattern allows only a single object-instance of a class to ever exist during an application run.

There are two requirements to make a class adhere to the singleton pattern:

- Declaring the constructor of a class `private`. When you declare the `Superman` class's constructor `private` then the constructor isn't visible outside the class or in its subclasses. Only the instance and static methods of the `Superman` class are able to access the constructor and create instances of the `Superman` class.
- The second trick is to create a public static method usually named `getInstance()` to return the only instance. We create a private static object of the class `Superman` and return it via the `getInstance()` method. We can control when to instantiate the lone static private instance. Here's what the code looks like:

```

public class SupermanNaiveButCorrect {

    // We are initializing the object inline
    private static SupermanNaiveButCorrect superman = new SupermanNaiveBut
Correct();

    // We have marked the constructor private
    private SupermanNaiveButCorrect() {
    }

    public static SupermanNaiveButCorrect getInstance() {
        return superman;
    }

    // Object method
    public void fly() {
        System.out.println("I am Superman & I can fly !");
    }
}

```

Here's what your interviewer will tell you when you write this code:

- What if the no one likes Superman and instead creates Batman in the game. You just created Superman and he kept waiting without ever being called upon to save the world. It's a waste of Superman's time and also the memory and other resources he'll consume.
- What if the creation of Superman is a very resource-intensive effort after all he's coming from planet Krypton. We would really like to only create Superman once we need him. Or in programming-speak, we want to ***lazily initialize*** Superman

The next version is what most candidates would write and is incorrect.

```

public class SupermanWithFlaws {

    private static SupermanWithFlaws superman;

    private SupermanWithFlaws() {
    }

    // This will fail with multiple threads
    public static SupermanWithFlaws getInstance() {
        if (superman == null) {
            // A thread can be context switched at this point and
            // superman will evaluate to null for any other threads
            // testing the if condition. Now multiple threads will
            // fall into this if clause till the superman object is
            // assigned a value. All these threads will intialize the
            // superman object when it should have been initialized
            // only one.
            superman = new SupermanWithFlaws();
        }
    }
}

```

```

        return superman;
    }

    // Object method
    public void fly() {
        System.out.println("I am Superman & I can fly !");
    }
}

```

As any reader of this course should realize by now (if I have done a good job of teaching) that the `getInstance()` method would fail miserably in a multi-threaded scenario. A thread can context switch out just before it initializes the Superman, causing later threads to also fall into the if clause and end up creating multiple superman objects.

The naive way to fix this issue is to use our good friend `synchronized` and either add `synchronized` to the signature of the `getInstance()` method or add a `synchronized` block within the method body. Thee mutual exclusion ensures that only one thread gets to initialize the object.

```

public class SupermanCorrectButSlow {

    private static SupermanCorrectButSlow superman;

    private SupermanCorrectButSlow() {

    }

    public static SupermanCorrectButSlow getInstance() {
        synchronized(Superman.class) {
            if (superman == null) {
                superman = new SupermanWithFlaws();
            }
        }
        return superman;
    }

    // Object method
    public void fly() {
        System.out.println("I am Superman & I can fly !");
    }
}

```

The con of the above solution is that every invocation of the `getInstance()` method causes the invoking thread to synchronize, which is prohibitively more expensive in terms of performance than non-synchronized snippets of code. Can we synchronize only when initializing the singleton instance and not at other times? The answer is yes and leads us to an implementation known as **double checked locking**. The idea is that we do two checks for `superman == null` in a nested fashion. The first check is without synchronization and the second with. Once a singleton instance has been initialized, all future invocations of

the `getInstance()` method don't pass the first null check and return the instance without getting involved in synchronization. Effectively, threads only synchronize when the singleton instance has not yet been initialized.

```
public class SupermanSlightlyBetter {

    private static SupermanSlightlyBetter superman;

    private SupermanSlightlyBetter() {

    }

    public static SupermanSlightlyBetter getInstance() {

        // Check if object is uninitialized
        if (superman == null) {

            // Now synchronize on the class object, so that only
            // 1 thread gets a chance to initialize the superman
            // object. Note that multiple threads can actually find
            // the superman object to be null and fall into the
            // first if clause
            synchronized (SupermanSlightlyBetter.class) {

                // Must check once more if the superman object is still
                // null. It is possible that another thread might have
                // initialized it already as multiple threads could have
                // made past the first if check.
                if (superman == null) {
                    superman = new SupermanSlightlyBetter();
                }
            }
        }
        return superman;
    }
}
```

The above solution seems almost correct. In fact, it'll appear correct unless you understand how the intricacies of Java's memory model and compiler optimizations can affect thread behaviors. The memory model defines what state a thread may see when it reads a memory location modified by other threads. The above solution needs one last missing piece but before we add that consider the below scenario:

1. Thread A comes along and gets to the second if check and allocates memory for the `superman` object but doesn't complete construction of the object and gets switched out. The Java memory model doesn't ensure that the constructor completes before the reference to the new object is assigned to an instance. It is possible that the variable `superman` is non-null but the object it points to, is still being initialized in the constructor by another thread.

2. Thread B wants to use the superman object and since the memory is already allocated for the object it fails the first if check and returns a semi-constructed superman object. Attempt to use a partially created object results in a crash or undefined behavior.

To fix the above issue, we mark our `superman` static object as `volatile`. The *happens-before* semantics of `volatile` guarantee that the faulty scenario of threads A and B never happens.

By now you'll probably appreciate how hard it is to get the singleton pattern right in a multithreaded scenario. Also, note that the discussed solution works with Java 1.5 or above. `volatile`'s behavior in Java 1.4 and earlier is different and the **double checked locking** pattern is broken when run on those versions of Java.

Last but not the least, double-checked locking (DCL) is an antipattern and its utility has dwindled over time as the JVM startup and uncontended synchronization speeds have improved.

The next lesson explains alternate Singleton implementations in Java.

Complete Code

```
public class Superman {  
    private static volatile Superman superman;  
    private Superman() {}  
  
    public static Superman getInstance() {  
        if (superman == null) {  
            synchronized (Superman.class) {  
                if (superman == null) {  
                    superman = new Superman();  
                }  
            }  
        }  
        return superman;  
    }  
  
    public void fly() {  
        System.out.println("I am Superman & I can fly !");  
    }  
}
```

... continued

This lesson continues the discussion on implementing the Singleton pattern in Java.

Implementing a thread-safe Singleton class is a popular interview question with the double checked locking as the most debated implementation. For completeness, we present below all the various evolutions of the Singleton pattern in Java.

- The easiest way to create a singleton is to mark the constructor of the class private and create a private static instance of the class that is initialized inline. The instance is returned through a public getter method. The drawback of this approach is if the singleton object is never used then we have spent resources creating and retaining the object in memory. The static member is initialized when the class is loaded. Additionally, the singleton instance can be expensive to create and we may want to delay creating the object till it is actually required.

Singleton eager initialization#

```
public class Superman {  
    private static final Superman superman = new Superman();  
  
    private Superman() {}  
  
    public static Superman getInstance() {  
        return superman;  
    }  
}
```

A variant of the same approach is to initialize the instance in a static block.

Singleton initialization in static block#

```
public class Superman {  
    private static Superman superman;  
  
    static {  
        try {  
            superman = new Superman();  
        } catch (Exception e) {  
            // Handle exception here  
        }  
    }  
}
```

```

private Superman() {
}

public static Superman getInstance() {
    return superman;
}
}

```

The above approach is known as Eager Initialization because the singleton is initialized irrespective of whether it is used or not. Also, note that we don't need any explicit thread synchronization because it is provided for free by the JVM when it loads the **Superman** class.

```

class Demonstration {
    public static void main( String args[] ) {
        Superman superman = Superman.getInstance();
        superman.fly();
    }
}

class Superman {
    private static Superman superman = new Superman();

    private Superman() {
    }

    public static Superman getInstance() {
        return superman;
    }

    public void fly() {
        System.out.println("I am flyyyyyinggggg ...");
    }
}

```

- The next approach is to lazily create the singleton object. Lazy initialization means delaying creating a resource till the time of its first use. This saves precious resources if the singleton object is never used or is expensive to create. First, let's see how the pattern will be implemented in a single threaded environment.

Singleton lazy initialization in static block#

```

public class Superman {
    private static Superman superman;
}

```

```

private Superman() {
}

public static Superman getInstance() {
    if (superman == null) {
        superman = new Superman();
    }
    return superman;
}
}

```

With the above approach we are able to introduce lazy initialization, however, the class isn't thread-safe. Also, we needlessly check if the instance is null every time we invoke `getInstance()` method.

To make the above code thread safe we synchronize the `getInstance()` method and get a thread-safe class.

Thread safe#

```

public class Superman {
    private static Superman superman;

    private Superman() {
    }

    public synchronized static Superman getInstance() {
        if (superman == null) {
            superman = new Superman();
        }
        return superman;
    }
}

```

Note that the method is synchronized on the class object. The problem with the above approach is we are serializing access for threads even after the singleton object is safely initialized the first time. This slows down performance unnecessarily. The next evolution is to move the lock inside of the method.

```

class Demonstration {
    public static void main( String args[] ) {
        Superman superman = Superman.getInstance();
        superman.fly();

    }
}

```

```
class Superman {  
    private static Superman superman;  
  
    private Superman() {  
    }  
  
    public synchronized static Superman getInstance() {  
  
        if (superman == null) {  
            superman = new Superman();  
        }  
  
        return superman;  
    }  
  
    public void fly() {  
        System.out.println("I am flyyyyyinggggg ...");  
    }  
}  
  
class Demonstration {  
    public static void main( String args[] ) {  
        Superman superman = Superman.getInstance();  
        superman.fly();  
  
    }  
}  
  
class Superman {  
    private static Superman superman;  
  
    private Superman() {  
    }  
  
    public synchronized static Superman getInstance() {  
  
        if (superman == null) {  
            superman = new Superman();  
        }  
  
        return superman;  
    }  
  
    public void fly() {  
        System.out.println("I am flyyyyyinggggg ...");  
    }  
}  
class Demonstration {
```

```

public static void main( String args[] ) {
    Superman superman = Superman.getInstance();
    superman.fly();
}
}

class Superman {

    private Superman() {
    }

    private static class Holder {
        private static final Superman superman = new Superman();
    }

    public static Superman getInstance() {
        return Holder.superman;
    }

    public void fly() {
        System.out.println("I am flyyyyyinggggg ...");
    }
}

```

Multithreaded Merge Sort

Learn how to perform Merge sort using threads.

Merge Sort#

Merge sort is a typical text-book example of a recursive algorithm and the poster-child of the divide and conquer strategy. The idea is very simple, we divide the array into two equal parts, sort them recursively and then combine the two sorted arrays. The base case for recursion occurs when the size of the array reaches a single element. An array consisting of a single element is already sorted.

The running time for a recursive solution is expressed as a *recurrence equation*. An equation or inequality that describes a function in terms of its own value on smaller inputs is called a recurrence equation. The running time for a recursive algorithm is the solution to the recurrence equation. The recurrence equation for recursive algorithms usually takes on the following form:

Running Time = Cost to divide into n subproblems + n * Cost to solve each of the n problems + Cost to merge all n problems

In the case of merge sort, we divide the given array into two arrays of equal size, i.e. we divide the original problem into sub-problems to be solved recursively.

Following is the recurrence equation for merge sort.

Running Time = Cost to divide into 2 unsorted arrays + 2 * Cost to sort half the original array + Cost to merge 2 sorted arrays

$$\begin{aligned}
 T(n) &= \text{Cost to divide into } 2 \text{ unsorted arrays} + 2 * \\
 T\left(\frac{n}{2}\right) &+ \text{Cost to merge } 2 \text{ sorted arrays when } n > 1 \\
 T(n) &= \text{Cost to divide into } 2 \text{ unsorted arrays} + 2 * T(2n) \\
 &+ \text{Cost to merge } 2 \text{ sorted arrays when } n > 1 \\
 T(n) &= O(1) \text{ when } n = 1 \quad T(n) = O(1) \text{ when } n = 1
 \end{aligned}$$

Remember the *solution* to the recurrence equation will be the *running time* of the algorithm on an input of size n . Without getting into the details of how we'll solve the recurrence equation, the running time of merge sort is

$$O(n \lg n)$$

where n is the size of the input array.

Merge sort lends itself very nicely for parallelism. Note that the subdivided problems or subarrays don't overlap with each other so each thread can work on its assigned subarray without worrying about synchronization with other threads. There is no data or state being shared between threads. There's only one caveat, we need to make sure that peer threads at each level of recursion finish before we attempt to merge the subproblems.

Let's first implement the single threaded version of Merge Sort and then attempt to make it multithreaded. Note that merge sort can be implemented without using extra space but the implementation becomes complex so we'll allow ourselves the luxury of using extra space and stick to a simple-to-follow implementation.

```

1. class SingleThreadedMergeSort {
2.
3.     private static int[] scratch = new int[10];
4.
5.     public static void main( String args[] ) {
6.         int[] input = new int[]{ 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 }
7.         ;
8.         printArray(input, "Before: ");
9.         mergeSort(0, input.length-1, input);
10.    }
11. }
```

```
9.     printArray(input, "After:  ");
10.    }
11. }
12.
13. private static void mergeSort(int start, int end, int[
] input) {
14.
15.     if (start == end) {
16.         return;
17.     }
18.
19.     int mid = start + ((end - start) / 2);
21.
22.     // sort first half
23.     mergeSort(start, mid, input);
24.
25.     // sort second half
26.     mergeSort(mid + 1, end, input);
27.
28.     // merge the two sorted arrays
29.     int i = start;
30.     int j = mid + 1;
31.     int k;
32.
33.     for (k = start; k <= end; k++) {
34.         scratch[k] = input[k];
35.     }
36.
37.     k = start;
38.     while (k <= end) {
39.
40.         if (i <= mid && j <= end) {
41.             input[k] = Math.min(scratch[i], scratch[j]);
42.
43.             if (input[k] == scratch[i]) {
44.                 i++;
45.             } else {
46.                 j++;
47.             }
48.         } else if (i <= mid && j > end) {
49.             input[k] = scratch[i];
50.             i++;
51.         } else {
52.             input[k] = scratch[j];
53.             j++;
54.         }
55.         k++;
56.     }
57. }
```

```

private static void printArray(int[] input, String msg) {
    System.out.println();
    System.out.print(msg + " ");
    for (int i = 0; i < input.length; i++) {
        System.out.print(" " + input[i] + " ");
    }
    System.out.println();
}
}

```

In the above single threaded code, the opportunity to parallelize the processing of each sub-problem exists on **line 23** and **line 26**. We create two threads and allow them to carry on processing the two subproblems. When both are done, then we combine the solutions. Note that the threads work on the same array but on completely exclusive portions of it, there's no chance of synchronization issues coming up.

Below is the multithreaded code for Merge sort. Note the code is slightly different than the single threaded version to account for changes required for concurrent code.

```

import java.util.Random;

class Demonstration {

    private static int SIZE = 25;
    private static Random random = new Random(System.currentTimeMillis());
    private static int[] input = new int[SIZE];

    static private void createTestData() {
        for (int i = 0; i < SIZE; i++) {
            input[i] = random.nextInt(10000);
        }
    }

    static private void printArray(int[] input) {
        System.out.println();
        for (int i = 0; i < input.length; i++) {
            System.out.print(" " + input[i] + " ");
        }
        System.out.println();
    }

    public static void main( String args[] ) {
        createTestData();
    }
}

```

```

        System.out.println("Unsorted Array");
        printArray(input);
        long start = System.currentTimeMillis();
        (new MultiThreadedMergeSort()).mergeSort(0, input.length - 1, input);
        long end = System.currentTimeMillis();
        System.out.println("\n\nTime taken to sort = " + (end - start) + " milliseconds");
        System.out.println("Sorted Array");
        printArray(input);
    }
}

class MultiThreadedMergeSort {

    private static int SIZE = 25;
    private int[] input = new int[SIZE];
    private int[] scratch = new int[SIZE];

    void mergeSort(final int start, final int end, final int[] input) {

        if (start == end) {
            return;
        }

        final int mid = start + ((end - start) / 2);

        // sort first half
        Thread worker1 = new Thread(new Runnable() {

            public void run() {
                mergeSort(start, mid, input);
            }
        });

        // sort second half
        Thread worker2 = new Thread(new Runnable() {

            public void run() {
                mergeSort(mid + 1, end, input);
            }
        });

        // start the threads
        worker1.start();
        worker2.start();

        try {
            worker1.join();

```

```

        worker2.join();
    } catch (InterruptedException ie) {
        // swallow
    }

    // merge the two sorted arrays
    int i = start;
    int j = mid + 1;
    int k;

    for (k = start; k <= end; k++) {
        scratch[k] = input[k];
    }

    k = start;
    while (k <= end) {

        if (i <= mid && j <= end) {
            input[k] = Math.min(scratch[i], scratch[j]);

            if (input[k] == scratch[i]) {
                i++;
            } else {
                j++;
            }
        } else if (i <= mid && j > end) {
            input[k] = scratch[i];
            i++;
        } else {
            input[k] = scratch[j];
            j++;
        }
        k++;
    }
}
}

```

We create two threads on lines **51** and **59** and then wait for them to finish on **lines 67-68**. On smaller datasets the speed-up achieved may not be visible but larger datasets which are processed on multiprocessor machines, the speed-up effect will be much more pronounced.

Asynchronous to Synchronous Problem

A real-life interview question asking to convert asynchronous execution to synchronous execution.

Problem Statement

This is an actual interview question asked at Netflix.

Imagine we have an `Executor` class that performs some useful task asynchronously via the method `asynchronousExecution()`. In addition the method accepts a callback object which implements the `Callback` interface. the object's `done()` gets invoked when the asynchronous execution is done. The definition for the involved classes is below:

Executor Class#

```
public class Executor {  
    public void asynchronousExecution(Callback callback) throws Exception {  
        Thread t = new Thread(() -> {  
            // Do some useful work  
            try {  
                // Simulate useful work by sleeping for 5 seconds  
                Thread.sleep(5000);  
            } catch (InterruptedException ie) {  
            }  
            callback.done();  
        });  
        t.start();  
    }  
}
```

Callback Interface#

```
public interface Callback {  
    public void done();  
}
```

An example run would be as follows:

```
class Demonstration {  
    public static void main( String args[] ) throws Exception {  
        Executor executor = new Executor();  
        executor.asynchronousExecution(() -> {  
            System.out.println("I am done");  
        });  
  
        System.out.println("main thread exiting...");  
    }  
}
```

```

interface Callback {

    public void done();
}

class Executor {

    public void asynchronousExecution(Callback callback) throws Exception {

        Thread t = new Thread(() -> {
            // Do some useful work
            try {
                Thread.sleep(5000);
            } catch (InterruptedException ie) {
            }
            callback.done();
        });
        t.start();
    }
}

```

Note how the main thread exits before the asynchronous execution is completed.

Your task is to make the execution synchronous without changing the original classes (imagine, you are given the binaries and not the source code) so that main thread waits till asynchronous execution is complete. In other words, the highlighted **line#8** only executes once the asynchronous task is complete.

Solution#

The problem here asks us to convert asynchronous code to synchronous code without modifying the original code. The requirement that the main thread should block, till the asynchronous execution is complete hints at using some kind of notification/signalling mechanism. The main thread *waits* on something, which is then *signaled* by the asynchronous execution thread. Semaphore is the first thought that may come to your mind for solving this problem. However, I was told to use primitive Java synchronization constructs i.e. `notify()` and `wait()` methods on the `Object` class.

Since we can't modify the original code, we'll extend a new class `SynchronousExecutor` from the given `Executor` class and override the `asynchronousExecution()` method. The trick here is to invoke the original

asynchronous implementation using `super.asynchronousExecution()` inside the overridden method. The overridden method would look like:

```
public void asynchronousExecution(Callback callback) throws Exception {  
    // Pass something to the base class's asynchronous  
    // method implementation that the base class can notify on  
    // Call the asynchronous executor  
    super.asynchronousExecution(callback);  
    // Wait on something that the base class's asynchronous  
    // method implementation notifies for  
}
```

Next, we can create an object for notification and waiting purposes

```
public void asynchronousExecution(Callback callback) throws Exception {  
    Object signal = new Object();  
    // Call the asynchronous executor  
    super.asynchronousExecution(callback);  
    synchronized (signal){  
        signal();  
    }  
}
```

Now we need to pass the `signal` object to the superclass's `asynchronousExecution()` method so that the asynchronous execution thread can `notify()` the `signal` variable once asynchronous execution is complete. We pass in the `callback` object to the super class's method. We can wrap the original callback in another callback object and pass also in our `signal` variable to the super class. Let's see how we can achieve that:

```
public void asynchronousExecution(Callback callback) throws Exception {  
  
    Object signal = new Object();  
    Callback cb = new Callback() {  
        @Override  
        public void done() {  
            callback.done();  
            synchronized (signal) {  
                signal.notify();  
            }  
        }  
    };  
    // Call the asynchronous executor  
    super.asynchronousExecution(cb);  
    synchronized (signal) {  
        signal.wait();  
    }  
}
```

Note that the variable `signal` gets *captured* in the scope of the new callback that we define. However, the captured variable must be defined `final` or be effectively `final`. Since we are assigning the variable only once, it is effectively `final`. The code so far defines the basic structure of the solution and we need to add a few missing pieces for it to work.

Remember we can't use `wait()` method without enclosing it inside a while loop as supurious wakeups can occur. Let's fix that

```
public void asynchronousExecution(Callback callback) throws Exception {
    Object signal = new Object();
    boolean isDone = false;
    Callback cb = new Callback() {
        @Override
        public void done() {
            callback.done();
            synchronized (signal) {
                signal.notify();
                isDone = true;
            }
        }
    };
    // Call the asynchronous executor
    super.asynchronousExecution(cb);
    synchronized (signal) {
        while (!isDone) {
            signal.wait();
        }
    }
}
```

Note that the invariant here is `isDone` which is set to true after the asynchronous execution is complete. The last problem here is that `isDone` isn't `final`. We can't declare it final because `isDone` gets assigned to after initialization. At this a slightly less elegant but workable solution is to use a boolean array of size 1 to represent our boolean. The array can be final because it gets assigned memory at initialization but the contents of the array can be changed later without compromising the finality of the variable.

```
@Override
public void asynchronousExecution(Callback callback) throws Exception {
    Object signal = new Object();
    final boolean[] isDone = new boolean[1];
    Callback cb = new Callback() {
        @Override
        public void done() {
            callback.done();
            synchronized (signal) {
                signal.notify();
                isDone[0] = true;
            }
        }
    }
```

```
};

// Call the asynchronous executor
super.asynchronousExecution(cb);
synchronized (signal) {
    while (!isDone[0]) {
        signal.wait();
    }
}
```

Complete Code#

The complete code appears below:

```
class Demonstration {
    public static void main( String args[] ) throws Exception {
        SynchronousExecutor executor = new SynchronousExecutor();
        executor.asynchronousExecution(() -> {
            System.out.println("I am done");
        });

        System.out.println("main thread exiting...");
    }
}

interface Callback {

    public void done();
}

class SynchronousExecutor extends Executor {

    @Override
    public void asynchronousExecution(Callback callback) throws Exception {

        Object signal = new Object();
        final boolean[] isDone = new boolean[1];

        Callback cb = new Callback() {

            @Override
            public void done() {
                callback.done();
                synchronized (signal) {
                    signal.notify();
                    isDone[0] = true;
                }
            }
        };
    }
}
```

```

};

// Call the asynchronous executor
super.asynchronousExecution(cb);

synchronized (signal) {
    while (!isDone[0]) {
        signal.wait();
    }
}

}

class Executor {

    public void asynchronousExecution(Callback callback) throws Exception {

        Thread t = new Thread(() -> {
            // Do some useful work
            try {
                Thread.sleep(5000);
            } catch (InterruptedException ie) {
            }
            callback.done();
        });
        t.start();
    }
}

```

Note the main thread has its print-statement printed after the asynchronous execution thread print its print-statement verifying that the execution is now synchronous.

Follow-up Question: Is the method `asynchronousExecution()` thread-safe?

The way we have constructed the logic, all the variables in the overridden method will be created on the thread-stack for each thread therefore the method is threadsafe and multiple threads can execute it in parallel.

Nonblocking Stack

Problem#

Design a stack that doesn't use locks or `synchronized` and is thread-safe. You may assume that you are provided with an application-level API that mocks the hardware instruction compare-and-swap, to atomically compare and swap values at a memory location.

Solution#

This is a slightly advanced question that requires knowledge about atomic classes in Java. You can read the section on Atomics in this course and then come back to this question. Very briefly, a stack is a data structure that implements first-in, last-out semantics for stored items.

Usually, we want to atomically execute steps that require checking a value and then taking action based on the observed value. This is usually expressed as an `if-clause`. For instance, in the stack problem we would have a variable `top` that points to the top of the stack. Initially, this variable is `null` and in a single-threaded environment we would perform the following check when pushing the first element in the stack.

```
if (top == null) {
    top = new StackNode();
    // ... other initialization steps
}
```

As you can see, the above snippet is a check-then-act scenario - We check if `top` is already `null`, if so we initialize it to an object of `StackNode`. The above code fails in a multi-threaded environment because the check-then-act sequence can't be performed atomically. The straightforward path to make the code thread-safe is to either use locks or simply mark the methods `synchronized`. The code for a thread-safe stack using `synchronized` is presented below for context and before we delve into a solution without locking.

Main.java:

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

class Demonstration {

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

        SynchronizedStack<Integer> stackOfInts = new SynchronizedStack<>();
        ExecutorService executorService = Executors.newFixedThreadPool(20);
        int numThreads = 2;
        CyclicBarrier barrier = new CyclicBarrier(numThreads);
```

```

Integer testValue = new Integer(51);

try {
    for (int i = 0; i < numThreads; i++) {
        executorService.submit(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10000; i++) {
                    stackOfInts.push(testValue);
                }
            }

            try {
                barrier.await();
            } catch (InterruptedException | BrokenBarrierException ex) {
                System.out.println("ignoring exception");
                //ignore both exceptions
            }
        }
        for (int i = 0; i < 10000; i++) {
            stackOfInts.pop();
        }
    });
}
} finally {
    executorService.shutdown();
    executorService.awaitTermination(1, TimeUnit.HOURS);
}
}
}
}

```

StackNode.java

```

class StackNode<T> {
    private T item;
    private StackNode<T> next;

    public StackNode(T item) {
        this.item = item;
    }

    public StackNode<T> getNext() {
        return next;
    }

    public void setNext(StackNode<T> stackNode) {
        next = stackNode;
    }
}

```

```

public T getItem() {
    return this.item;
}
}

SynchronizedStack.java
class SynchronizedStack<T> {

    StackNode<T> top;

    public synchronized void push(T item) {

        if (top == null) {
            top = new StackNode<>(item);
        } else {
            StackNode<T> oldHead = top;
            top = new StackNode<>(item);
            top.setNext(oldHead);
        }
    }

    public synchronized T pop() {

        if (top == null) {
            return null;
        }

        StackNode<T> oldHead = top;
        top = top.getNext();
        return oldHead.getItem();
    }
}

```

There are several reasons to avoid locks which we described in detail in the Atomics section. The crux is that locking is a heavyweight synchronization mechanism that incurs significant overhead. In low to moderate contention situations, it may be better to execute critical sections of code speculatively in the hope that no other thread interferes. If interference is detected then actions are retried.

In interviews, this question can be solved either using the [AtomicReference](#) class or assuming that an API is provided that simulates the compare-and-swap (CAS) processor instruction. We'll first solve the problem using the mocked CAS API and then using [AtomicReference](#).

CAS-based solution#

Please read the CAS section in the Atomic Classes lesson before proceeding forward. We'll use the following class [SimulatedCompareAndSwap](#) to simulate the CAS processor instruction. Particularly, the method [compareAndSet\(\)](#) can be used to determine if the check-then-act steps were successfully executed atomically or not. Take some time to go through the listing of [SimulatedCompareAndSwap](#) and its associated tests in the code widget below:

Main.java:

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

class Demonstration {

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

        ExecutorService executorService = Executors.newFixedThreadPool(20);
        int numThreads = 7;
        // create an instance of simulated CAS instruction, that holds an integer value that is
        // simultaneously manipulated by 7 threads.
        final SimulatedCompareAndSwap<Integer> sharedInteger = new
        SimulatedCompareAndSwap<>(0);

        try {
            for (int i = 0; i < numThreads; i++) {
                executorService.submit(new Runnable() {
                    @Override
                    public void run() {

                        int expectedValue;
                        int newValue;
                        int updateAttempt = 0;

                        do {
                            expectedValue = sharedInteger.getValue();
                            newValue = expectedValue + 1;
                            System.out.println(Thread.currentThread().getName() + " update attempt
number # " + updateAttempt);
                            updateAttempt++;
                        } while (!sharedInteger.compareAndSet(expectedValue, newValue));
                    }
                });
            }
        } finally {
            executorService.shutdown();
            executorService.awaitTermination(1, TimeUnit.HOURS);
        }
    }
}
```

```

        System.out.println("Final integer value = " + sharedInteger.getValue());
    }
}

SimulatedCompareAndSwap.java
public class SimulatedCompareAndSwap<T> {

    private T value;

    // constructor to initialize the value
    public SimulatedCompareAndSwap(T initialValue) {
        value = initialValue;
    }

    synchronized T getValue() {
        return value;
    }

    synchronized T compareAndSwap(T expectedValue, T newValue) {

        if (value == null) {
            if (expectedValue == null) {
                value = newValue;
            }
            return null;
        }

        if (value.equals(expectedValue)) {
            value = newValue;
            return expectedValue;
        }

        // return the current value
        return value;
    }

    // This method uses the compareAndSwap() method to indicate if the CAS
    // instruction completed successfully or not.
    synchronized boolean compareAndSet(T expectedValue, T newValue) {
        T returnVal = compareAndSwap(expectedValue, newValue);

        if (returnVal == null && expectedValue == null) return true;
        else if (returnVal == null && expectedValue != null) return false;
        else {
            return returnVal.equals(expectedValue);
        }
    }
}

```

The above program shows 7 threads attempting to change the value of a shared integer value that is stored in an instance of `SimulatedCompareAndSwap`. Each thread spins in a loop until it is able to update the integer value.

The `SimulatedCompareAndSwap` is a generic class and an instance of the class stores value for the type argument, which is an integer in the above widget. Consider the following snippet:

```
SimulatedCompareAndSwap<Integer> myInt = new SimulatedCompareAndSwap<>(5);
```

The instance of `SimulatedCompareAndSwap` is storing an integer object which has a value of 5. Even though we are passing an `int` literal in the constructor but value is implicitly converted into an `Integer` object. Next, we can use attempt to change the value as follows:

```
SimulatedCompareAndSwap<Integer> myInt = new SimulatedCompareAndSwap<>(5);

// attempt to change the value when the expected value is incorrect
System.out.println(myInt.compareAndSet(5, 9));

// attempt to change the value with correct expected value
System.out.println(myInt.compareAndSet(5, 9));

// retrieve the current value
System.out.println(myInt.getValue());
```

The act of checking the current value of the typed parameter stored in the instance of `SimulatedCompareAndSwap` and then updating it to a new value is executed atomically using the method `compareAndSet()`. If the execution is successful a true boolean value is returned. However, it is possible that another thread reads and updates the stored value before the main thread has a chance to execute `compareAndSet()`. In that instance, the expected value parameter passed in by the main thread will not match the currently stored value and the attempt to change the stored value will be rejected, indicated by a false boolean return value.

When implementing a stack using `SimulatedCompareAndSwap` we'll store the current value of the `top` of the stack in the `SimulatedCompareAndSwap` instance. The trick is to let a thread read the current value of the `top` of stack and then attempt to change it using `compareAndSet()`. If the attempt is unsuccessful we simply re-read the `top` value again, since the value we previously had didn't match the expected value and try again in a loop until the operation succeeds. Let's see how the push operation for the stack will be implemented using this strategy.

```

do {
    // retrieve the current value of top
    oldHead = simulatedCAS.getValue();
    // create a new StackNode for the passed-in item.
    newHead = new StackNode<>(item);
    // Adjust the pointer
    newHead.setNext(oldHead);
} while (!simulatedCAS.compareAndSet(oldHead, newHead));
// attempt to atomically check and update

```

The above code is a partial snippet from the complete code for the `push()` method. The intent is to demonstrate how we loop until the expected value matches the stored value and the new value is updated. Similarly, the `pop()` method uses a loop to remove the top of the stack and update it with the next element atomically. The partial snippet from the `pop()` is shown below:

```

do {
    // get the current top of the stack
    returnValue = simulatedCAS.getValue();
    // if the top is null then simply return null
    if (returnValue == null) return null;
    // compute the new top of stack
    newHead = returnValue.getNext();
} while (!simulatedCAS.compareAndSet(returnValue, newHead));
// attempt to update the new top of stack

```

Note, that these loops run a finite number of times. The unluckiest of threads will eventually have its expected value match the current value and the `compareAndSet()` method will succeed. The complete code for the stack class `CASBasedStack` appears below:

Main.java

```

import java.util.concurrent.*;

class Demonstration {

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

        CASBasedStack<Integer> stack = new CASBasedStack<>();
        ExecutorService executorService = Executors.newFixedThreadPool(20);
        int numThreads = 2;
        CyclicBarrier barrier = new CyclicBarrier(numThreads);

        Integer testValue = new Integer(51);

        try {
            for (int i = 0; i < numThreads; i++) {

```

```

executorService.submit(new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            stack.push(testValue);
        }

        try {
            barrier.await();
        } catch (InterruptedException | BrokenBarrierException ex) {
            System.out.println("ignoring exception");
            //ignore both exceptions
        }

        for (int i = 0; i < 10000; i++) {
            stack.pop();
        }
    }
});

} finally {
    executorService.shutdown();
    executorService.awaitTermination(1, TimeUnit.HOURS);
}
}

System.out.println("10,000 pushes and pops completed by 2 threads. Current top of stack
" + stack.pop());
}
}

```

StackNode.java

```

public class StackNode<T> {
    private T item;
    private StackNode<T> next;

    public StackNode(T item) {
        this.item = item;
    }

    public StackNode<T> getNext() {
        return next;
    }

    public void setNext(StackNode<T> stackNode) {
        next = stackNode;
    }
}

```

```

    public T getItem() {
        return this.item;
    }
}

CASBasedStack.java
public class CASBasedStack<T> {

    private SimulatedCompareAndSwap<StackNode<T>> simulatedCAS;

    public CASBasedStack() {
        simulatedCAS = new SimulatedCompareAndSwap<>(null);

    }

    public void push(T item) {

        StackNode<T> oldHead;
        StackNode<T> newHead;

        do {
            // retrieve the current value of top
            oldHead = simulatedCAS.getValue();
            // create a new StackNode for the passed-in item.
            newHead = new StackNode<>(item);
            // Adjust the pointer
            newHead.setNext(oldHead);
        } while (!simulatedCAS.compareAndSet(oldHead, newHead));
        // attempt to atomically check and update
    }

    public T pop() {

        StackNode<T> returnValue;
        StackNode<T> newHead;

        do {
            // get the current top of the stack
            returnValue = simulatedCAS.getValue();
            // if the top is null then simply return null
            if (returnValue == null) return null;
            // compute the new top of stack
            newHead = returnValue.getNext();
        } while (!simulatedCAS.compareAndSet(returnValue, newHead));
        // attempt to update the new top of stack

        return returnValue.getItem();
    }
}

```

SimulatedCompareAndSwap.java

```
public class SimulatedCompareAndSwap<T> {

    private T value;

    // constructor to initialize the value
    public SimulatedCompareAndSwap(T initialValue) {
        value = initialValue;
    }

    synchronized T getValue() {
        return value;
    }

    synchronized T compareAndSwap(T expectedValue, T newValue) {

        if (value == null) {
            if (expectedValue == null) {
                value = newValue;
            }
            return null;
        }

        if (value.equals(expectedValue)) {
            value = newValue;
            return expectedValue;
        }

        // return the current value
        return value;
    }

    // This method uses the compareAndSwap() method to indicate if the CAS
    // instruction completed successfully or not.
    synchronized boolean compareAndSet(T expectedValue, T newValue) {
        T returnVal = compareAndSwap(expectedValue, newValue);

        if (returnVal == null && expectedValue == null) return true;
        else if (returnVal == null && expectedValue != null) return false;
        else {
            return returnVal.equals(expectedValue);
        }
    }
}
```

AtomicReference solution#

In the previous solution we mocked the compare-and-swap processor instruction with the class `SimulatedCompareAndSwap`. Fortunately, Java provides classes that can be used to atomically execute compare-and-swap operations for a variety of types. Chief among them are `AtomicInteger`, `AtomicLong` and `AtomicReference`.

We discuss the `AtomicReference` class in detail here and suggest the reader to go through that lesson, if not already familiar with the `AtomicReference` class before proceeding forward.

The solution using the `AtomicReference` class will resemble the solution we implemented using the `SimulatedCompareAndSwap` class. The trick is to be able to atomically update the `top` of the stack and for that we'll store the current top of the stack, which is of type `StackNode`, in an instance of `AtomicReference`. The `push()` method's core loop looks like the following:

```
do {  
    oldTop = top.get();  
    newTop = new StackNode<>(newItem);  
    newTop.setNext(oldTop);  
} while (!top.compareAndSet(oldTop, newTop));
```

A thread reads the current `top` of the stack and creates a new `StackNode` as the new top of the stack. The `next` pointer of the new top is set to the old top and then the thread makes an attempt to update the `top` value. It does so in a loop until successful. The core loop for the `pop()` operation is shown below:

```
do {  
    oldTop = top.get();  
    if (oldTop == null) return null;  
    newTop = oldTop.getNext();  
} while (!top.compareAndSet(oldTop, newTop));
```

The complete code for the non-blocking stack using `AtomicReference` is shown below:

Main.java

```
import java.util.concurrent.*;  
  
class Demonstration {  
    public static void main( String args[] ) throws Exception {
```

```

NonblockingStack<Integer> stack = new NonblockingStack<>();
ExecutorService executorService = Executors.newFixedThreadPool(20);
int numThreads = 2;
CyclicBarrier barrier = new CyclicBarrier(numThreads);

long start = System.currentTimeMillis();
Integer testValue = new Integer(51);

try {
    for (int i = 0; i < numThreads; i++) {
        executorService.submit(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10000; i++) {
                    stack.push(testValue);
                }
            }

            try {
                barrier.await();
            } catch (InterruptedException | BrokenBarrierException ex) {
                System.out.println("ignoring exception");
                //ignore both exceptions
            }
        });
    }
} finally {
    executorService.shutdown();
    executorService.awaitTermination(1, TimeUnit.HOURS);
}

System.out.println("Number of elements in the stack = " + stack.size());
}
}

```

[NonblockingStack.java](#)

```

import java.util.concurrent.atomic.AtomicReference;
import java.util.concurrent.atomic.AtomicInteger;

class NonblockingStack<T> {

```

```

private AtomicInteger count = new AtomicInteger(0);
private AtomicReference<StackNode<T>> top = new AtomicReference<>();

public int size() {
    return count.get();
}

public void push(T newItem) {
    StackNode<T> oldTop;
    StackNode<T> newTop;
    do {
        oldTop = top.get();
        newTop = new StackNode<>(newItem);
        newTop.setNext(oldTop);
    } while (!top.compareAndSet(oldTop, newTop));

    count.incrementAndGet();
}

public T pop() {
    StackNode<T> oldTop;
    StackNode<T> newTop;

    do {
        oldTop = top.get();
        if (oldTop == null) return null;
        newTop = oldTop.getNext();
    } while (!top.compareAndSet(oldTop, newTop));

    count.decrementAndGet();
    return oldTop.getItem();
}
}

StackNode.java

```

```

public class StackNode<T> {
    private T item;
    private StackNode<T> next;

    public StackNode(T item) {
        this.item = item;
    }

    public StackNode<T> getNext() {
        return next;
    }
}

```

```
public void setNext(StackNode<T> stackNode) {  
    next = stackNode;  
}  
  
public T getItem() {  
    return this.item;  
}  
}
```

Epilogue

Closing remarks on the interview process.

Interviewing can be a stressful activity to undertake while juggling a full-time job and personal commitments. Rejections are hard to handle for most of us especially when they arrive in our inbox as canned messages from recruiters feigning sincerity with no meaningful insight into our interview performance. I tell candidates to go easy on themselves and not to let the judgments of five or more random strangers made in less than an hour, define for them their self-worth. Interviewing is after all an inexact science and I have seen far too many highly qualified candidates get rejected and many mediocre interview performances get rewarded with a hire. The key is never to give up and 'always be hustlin'. Honing one's interview skills and inculcating solid engineering talent will not let one go unnoticed for very long in an increasingly tech dominant economy with an insatiable demand for quality engineers !

Having said that, I am on a forever quest for self-improvement and receiving critique and feedback on my work. If you found the course lacking in value or have suggestions for improvements, I'd love to hear from you! Feel free to reach out to me on LinkedIn. Shoot me a connection request or just a message detailing what did and didn't work for you in the course. If there are any new topics or problems you'll like to see covered, I am all ears. Looking forward to hearing from you!

C. H. Afzal.

Every great product is a result of team-effort and so is this course. Collaborators on this course included the following good folks:

- [Ahsan Khalil](#) (Illustrations and graphic design)
- [Sana Bilal](#) (Content enhancement and development)

- [Maham Sana](#) (Content enhancement and development)
- Educative's Proofreading Ninjas

Last but not least, it is only human to err and so have I during the composition of this course. I am very grateful to folks who very kindly apprised me of the omissions and errors in the content and as a thank you note, I acknowledge them below.

- [Arun Shanmugam Kumar](#)
- [Sergey Lobov](#)
- [Andriy Tskitishvili](#)
- [Hanna Najjar](#)
- [Fahim ul Haq](#)
- [Bohan Zhang](#)
- [Manish Narula](#)
- [Stefan Cross](#)
- [Sanjeev Panday](#)
- [Diptanu Sarkar](#)
- [Chinmay Das](#)
- [Praneeth Nimmagadda](#)
- [Maria Komar](#)

Ordered Printing

This problem is about imposing an order on thread execution.

Problem Statement#

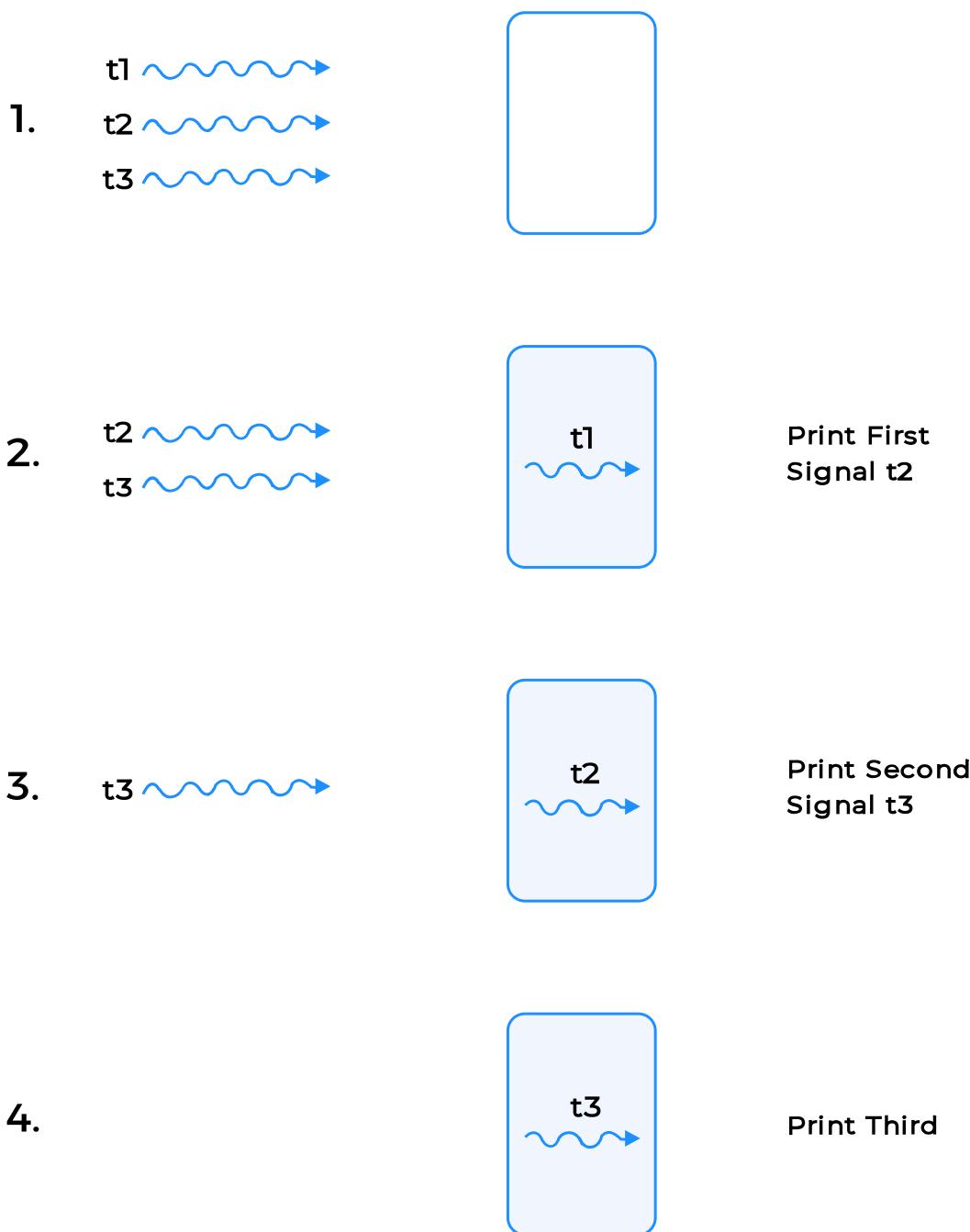
Suppose there are three threads t1, t2 and t3. t1 prints **First**, t2 prints **Second** and t3 prints **Third**. The code for the class is as follows:

```
public class OrderedPrinting {  
    public void printFirst() {  
        System.out.print("First");  
    }  
  
    public void printSecond() {  
        System.out.print("Second");  
    }  
  
    public void printThird() {  
        System.out.print("Third");  
    }  
}
```

Thread t1 calls printFirst(), thread t2 calls printSecond(), and thread t3 calls printThird(). The threads can run in any order. You have to synchronize the threads so that the functions **printFirst()**, **printSecond()** and **printThird()** are executed in order.

The workflow of the program is shown below:

Ordered Printing



Solution#

We present two solutions for this problem; one using the basic `wait()` & `notifyAll()` functions and the other using **CountDownLatch**.

Solution 1

In this solution, we have a class `OrderedPrinting` that consists of a private variable; `count`. The class consists of 3 functions `printFirst()`, `printSecond()` and `printThird()`. The structure of the class is as follows:

```
class OrderedPrinting {  
    int count;  
  
    public OrderedPrinting() {  
        count = 1;  
    }  
  
    public void printFirst() {  
    }  
  
    public void printSecond() {  
    }  
  
    public void printThird() {  
    }  
}
```

In the constructor, `count` is initialized with 1. Next we will explain the `printFirst()` function below:

```
public void printFirst() throws InterruptedException {  
  
    synchronized(this) {  
        System.out.println("First");  
        count++; //for printing Second, increment count  
        this.notifyAll();  
    }  
}
```

In `printFirst()`, "First" is printed. We do not need to check the value of `count` here. After printing, `count` is incremented for the next word to be printed. Any waiting threads are then notified via `notifyAll()`, signalling them to proceed.

```
public void printSecond() throws InterruptedException {  
  
    synchronized(this) {  
        while(count != 2) {  
            this.wait();  
        }  
        System.out.println("Second");  
        count++;  
        this.notifyAll();  
    }  
}
```

In the second method, the value of `count` is checked. If it is not equal to 2, the calling thread goes into wait. When the value of `count` reaches 2, the while loop is broken and "Second" is printed. The value of `count` is incremented for the next number to be printed and `notifyAll()` is called.

```
public void printThird() throws InterruptedException {  
    synchronized(this) {  
        while(count != 3) {  
            this.wait();  
        }  
        System.out.println("Third");  
    }  
}
```

The third method works in the same way as the second. The only difference being the check for `count` to be equal to 3. If it is, then "Third" is printed otherwise the calling thread waits.

To run our proposed solution, we will create another class to achieve multi-threading. When we extend `Thread` class, each of our thread creates a unique object and associates with the parent class. This class has two variables: one is the object of `OrderedPrinting` and the other is a string variable `method`. The string parameter checks the method to be invoked from `OrderedPrinting`.

```
class OrderedPrintingThread extends Thread {  
    private OrderedPrinting obj;  
    private String method;  
  
    public OrderedPrintingThread(OrderedPrinting obj, String method) {  
        this.method = method;  
        this.obj = obj;  
    }  
  
    public void run() {  
        //for printing "First"  
        if ("first".equals(method)) {  
            try {  
                obj.printFirst();  
            }  
            catch(InterruptedException e) {  
            }  
        }  
        //for printing "Second"  
        else if ("second".equals(method)) {  
            try {  
                obj.printSecond();  
            }  
            catch(InterruptedException e) {  
            }  
        }  
        //for printing "Third"  
    }  
}
```

```

        else if ("third".equals(method)) {
            try {
                obj.printThird();
            }
            catch(InterruptedException e) {
            }
        }
    }
}

```

We will be creating 3 threads in the `Main` class for testing each solution. Each thread will be passed the same object of `OrderedPrinting`. **t1** will call `printFirst()`, **t2** will call `printSecond()` and **t3** will call `printThird()`. The output shows printing done in the proper order i.e first, second and third irrespective of the calling order of threads.

```
class OrderedPrinting {
```

```

    int count;

    public OrderedPrinting() {
        count = 1;
    }

    public void printFirst() throws InterruptedException {
        synchronized(this){
            System.out.println("First");
            count++;
            this.notifyAll();
        }
    }

    public void printSecond() throws InterruptedException {
        synchronized(this){
            while(count != 2){
                this.wait();
            }
            System.out.println("Second");
            count++;
            this.notifyAll();
        }
    }

    public void printThird() throws InterruptedException {
        synchronized(this){
            while(count != 3){
                this.wait();
            }
        }
    }
}
```

```
        }
        System.out.println("Third");
    }

}

class OrderedPrintingThread extends Thread
{
    private OrderedPrinting obj;
    private String method;

    public OrderedPrintingThread(OrderedPrinting obj, String method)
    {
        this.method = method;
        this.obj = obj;
    }

    public void run()
    {
        //for printing "First"
        if ("first".equals(method))
        {
            try
            {
                obj.printFirst();
            }
            catch(InterruptedException e)
            {

            }
        }
        //for printing "Second"
        else if ("second".equals(method))
        {
            try
            {
                obj.printSecond();
            }
            catch(InterruptedException e)
            {

            }
        }
        //for printing "Third"
        else if ("third".equals(method))
        {
            try
            {
                obj.printThird();
            }
        }
    }
}
```

```

        }
    catch(InterruptedException e)
    {
        }

    }

}

public class Main
{
    public static void main(String[] args)
    {
        OrderedPrinting obj = new OrderedPrinting();

        OrderedPrintingThread t1 = new OrderedPrintingThread(obj, "first");
        OrderedPrintingThread t2 = new OrderedPrintingThread(obj, "second");
        OrderedPrintingThread t3 = new OrderedPrintingThread(obj, "third");

        t2.start();
        t3.start();
        t1.start();

    }
}

```

Solution using **CountDownLatch**#

The second solution includes the use of **CountDownLatch**; a synchronization utility used to achieve concurrency. It manages multithreading where a certain sequence of operations or tasks is required. Everytime a thread finishes its work, `countdown()` is invoked, decrementing the counter by 1. Once this count reaches zero, `await()` is notified and control is given back to the main thread that has been waiting for others to finish.

The basic structure of the class `OrderedPrinting` is the same as presented in solution 1 with the only difference of using **countdownlatch** instead of **volatile** variable. We have 2 **countdownlatch** variables that get initialized with 1 each.

```

class OrderedPrinting {
    CountDownLatch latch1;
    CountDownLatch latch2;

    public OrderedPrinting() {
        latch1 = new CountDownLatch(1);
        latch2 = new CountDownLatch(1);
    }
}

```

```
    }  
}
```

In `printFirst()` method, `latch1` decrements and reaches 0, waking up the waiting threads consequently. In `printSecond()`, if `latch1` is free (reached 0), then the printing is done and `latch2` is decremented. Similarly in the third method `printThird()`, `latch2` is checked and printing is done. The latches here act like switches/gates that get closed and opened for particular actions to pass.

```
public void printFirst() throws InterruptedException {  
    //print and notify waiting threads  
    System.out.println("First");  
    latch1.countDown();  
}  
  
public void printSecond() throws InterruptedException {  
    //wait if "First" has not been printed yet  
    latch1.await();  
    //print and notify waiting threads  
    System.out.println("Second");  
    latch2.countDown();  
}  
  
public void printThird() throws InterruptedException {  
    //wait if "Second" has not been printed yet  
    latch2.await();  
    System.out.println("Third");  
}
```

As in the previous solution, we create `OrderedPrintingThread` class which extends the `Thread` class. Details of this class are explained at length above.

```
import java.util.concurrent.CountDownLatch;  
  
class OrderedPrinting  
{  
    CountDownLatch latch1;  
    CountDownLatch latch2;  
  
    public OrderedPrinting()  
    {  
        latch1 = new CountDownLatch(1);  
        latch2 = new CountDownLatch(1);  
    }  
  
    public void printFirst() throws InterruptedException  
    {  
        System.out.println("First");  
        latch1.countDown();  
    }
```

```
}

public void printSecond() throws InterruptedException
{
    latch1.await();
    System.out.println("Second");
    latch2.countDown();
}

public void printThird() throws InterruptedException
{
    latch2.await();
    System.out.println("Third");
}

class OrderedPrintingThread extends Thread
{
    private OrderedPrinting obj;
    private String method;

    public OrderedPrintingThread(OrderedPrinting obj, String method)
    {
        this.method = method;
        this.obj = obj;
    }

    public void run()
    {
        if ("first".equals(method))
        {
            try
            {
                obj.printFirst();
            }
            catch(InterruptedException e)
            {

            }
        }
        else if ("second".equals(method))
        {
            try
            {
                obj.printSecond();
            }
            catch(InterruptedException e)
            {
            }
        }
    }
}
```

```

        }
    else if ("third".equals(method))
    {
        try
        {
            obj.printThird();
        }
        catch(InterruptedException e)
        {

        }
    }
}

public class Main
{
    public static void main(String[] args)
    {
        OrderedPrinting obj = new OrderedPrinting();

        OrderedPrintingThread t1 = new OrderedPrintingThread(obj, "first");
        OrderedPrintingThread t2 = new OrderedPrintingThread(obj, "second");
        OrderedPrintingThread t3 = new OrderedPrintingThread(obj, "third");

        t3.start();
        t2.start();
        t1.start();

    }
}

```

Printing Foo Bar n Times

Learn how to execute threads in a specific order for a user specified number of iterations.

Problem Statement#

Suppose there are two threads t1 and t2. t1 prints **Foo** and t2 prints **Bar**. You are required to write a program which takes a user input n. Then the two threads print Foo and Bar alternately n number of times. The code for the class is as follows:

```

class PrintFooBar {

    public void PrintFoo() {
        for (int i = 1 i <= n; i++){
            System.out.print("Foo");
        }
    }

    public void PrintBar() {
        for (int i = 1 i <= n; i++){
            System.out.print("Bar");
        }
    }
}

```

```

        }
    }

    public void PrintBar() {
        for (int i = 1; i <= n; i++) {
            System.out.print("Bar");
        }
    }
}

```

The two threads will run sequentially. You have to synchronize the two threads so that the functions PrintFoo() and PrintBar() are executed in an order. The workflow is shown below:

Solution#

We will solve this problem using the basic utilities of `wait()` and `notifyAll()` in Java. The basic structure of `FooBar` class is given below:

```

class FooBar {
    private int n;
    private int flag = 0;

    public FooBar(int n) {
        this.n = n;
    }

    public void foo() {
    }

    public void bar() {
}
}

```

Two private instances of the class are integers `n`, and `flag`.

`n` is the user input that tells how many times "Foo" and "Bar" should be printed. `flag` is an integer based on which the words are printed. When the value of `flag` is 0, the word "Foo" will be printed and it will be incremented. This way "Bar" can be printed next. `flag` is initialized with 0 because the printing has to start with "Foo". The class consists of two methods `foo()` and `bar()` and their structures are given below:

```

public void foo() {

    for (int i = 1; i <= n; i++) {
        synchronized(this) {
            while (flag == 1) {
}

```

```
        try {
            this.wait();
        }
        catch (Exception e) {
        }
    }
    System.out.print("Foo");
    flag = 1;
    this.notifyAll();
}
}
```

In `foo()`, a loop is iterated `n` (user input) number of times. For synchronization purpose, the printing operation is locked in `synchronize(this)` block. This is done to ensure proper sequence of printing. If `flag` is 0 then “Foo” is printed, then `flag` is set to 1 and any waiting threads are notified via `notifyAll()`. While the value of `flag` is 1, then `wait()` blocks the calling thread.

```
public void bar() {  
    for (int i = 1; i <= n; i++) {  
        synchronized(this) {  
            while (flag == 0) {  
                try {  
                    this.wait();  
                }  
                catch (Exception e) {  
                }  
            }  
            System.out.print("Bar");  
            flag = 0;  
            this.notifyAll();  
        }  
    }  
}
```

Similarly in `bar()`, the loop is iterated `n` times. In every iteration, the while loop checks if the value of `flag` is 0. If it is, then it means it is not yet bar's turn to be printed and the calling thread will `wait()`. When the value of `flag` changes to 1, the waiting thread can resume execution. "Bar" is printed and `flag` is changed to 0 for "Foo" to be printed next. All the waiting threads are then notified via `notifyAll()` that this thread has finished its work.

We will create a new class `FooBarThread` that extends Thread. This enables us to run `FooBar` methods in separate threads concurrently. The class consists of a `FooBar` object along with a string `method` which holds the name of the function to be called. If `method` matches "foo" then `fooBar.foo()` is called. If `method` matches "bar", then `fooBar.bar()` is called.

```

class FooBarThread extends Thread {

    FooBar fooBar;
    String method;

    public FooBarThread(FooBar fooBar, String method){
        this.fooBar = fooBar;
        this.method = method;
    }

    public void run() {
        if ("foo".equals(method)) {
            fooBar.foo();
        }
        else if ("bar".equals(method)) {
            fooBar.bar();
        }
    }
}

```

To test our code, We will create two threads; **t1** and **t2**. An object of `FooBar` is initialized with `3`. Both threads will be passed the same object of `FooBar`. **t1** calls `foo()` & **t2** calls `bar()`.

```

class FooBar {

    private int n;
    private int flag = 0;

    public FooBar(int n) {
        this.n = n;
    }

    public void foo() {

        for (int i = 1; i <= n; i++) {
            synchronized(this) {
                while (flag == 1) {
                    try {
                        this.wait();
                    }
                    catch (Exception e) {

                    }
                }
                System.out.print("Foo");
                flag = 1;
                this.notifyAll();
            }
        }
    }
}

```

```

public void bar() {
    for (int i = 1; i <= n; i++) {
        synchronized(this) {
            while (flag == 0) {
                try {
                    this.wait();
                }
                catch (Exception e) {
                }
            }
            System.out.println("Bar");
            flag = 0;
            this.notifyAll();
        }
    }
}

class FooBarThread extends Thread {
    FooBar fooBar;
    String method;

    public FooBarThread(FooBar fooBar, String method){
        this.fooBar = fooBar;
        this.method = method;
    }

    public void run() {
        if ("foo".equals(method)) {
            fooBar.foo();
        }
        else if ("bar".equals(method)) {
            fooBar.bar();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        FooBar fooBar = new FooBar(3);

        Thread t1 = new FooBarThread(fooBar,"foo");
        Thread t2 = new FooBarThread(fooBar,"bar");
    }
}

```

```

        t2.start();
        t1.start();

    }
}

```

Printing Number Series (Zero, Even, Odd)

This problem is about repeatedly executing threads which print a specific type of number. Another variation of this problem; print even and odd numbers; utilizes two threads instead of three.

Problem Statement#

Suppose we are given a number n based on which a program creates the series $010203...on$. There are three threads t_1 , t_2 and t_3 which print a specific type of number from the series. t_1 only prints zeros, t_2 prints odd numbers and t_3 prints even numbers from the series. The code for the class is given as follows:

```

class PrintNumberSeries {

    public PrintNumberSeries(int n) {
        this.n = n;
    }

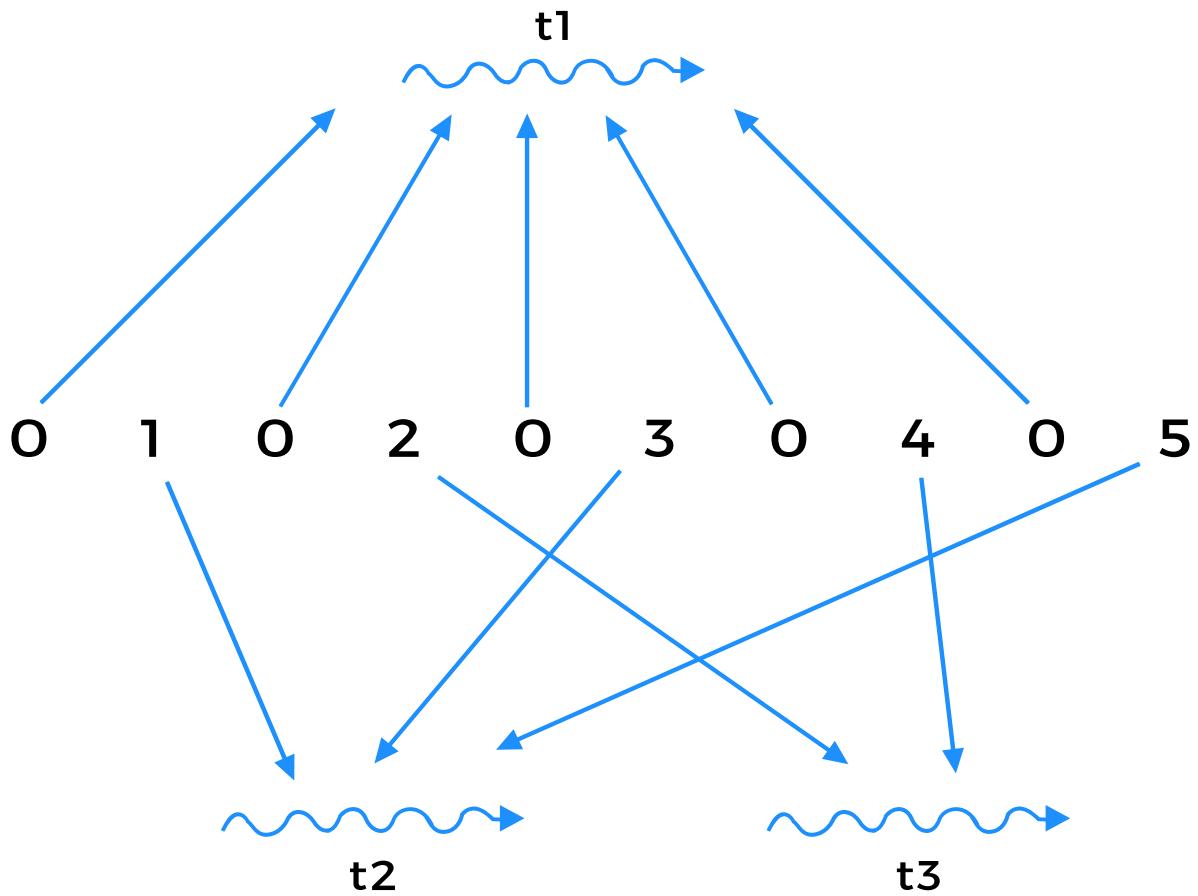
    public void PrintZero() {
    }
    public void PrintOdd() {
    }
    public void PrintEven() {
    }

}

```

You are required to write a program which takes a user input n and outputs the number series using three threads. The three threads work together to print zero, even and odd numbers. The threads should be synchronized so that the functions $\text{PrintZero}()$, $\text{PrintOdd}()$ and $\text{PrintEven}()$ are executed in an order.

The workflow of the program is shown below:



Solution#

This problem is solved by using Semaphores in Java. Semaphores are used to restrict the number of threads that can access some (physical or logical) resource at the same time. Our solution makes use of three semaphores; `zeroSem` for printing zeros, `oddSem` for printing odd numbers and `evenSem` for printing even numbers. The basic structure of the class is given below:

```
class PrintNumberSeries {

    private int n;
    private Semaphore zeroSem, oddSem, evenSem;

    public PrintNumberSeries(int n) {
    }

    public void PrintZero() {
    }

    public void PrintOdd() {
    }

    public void PrintEven() {
    }
}
```

```
}
```

`n` is the user input that prints the series till n th number. The constructor of this class appears below:

```
public PrintNumberSeries(int n) {  
    this.n = n;  
    zeroSem = new Semaphore(1);  
    oddSem = new Semaphore(0);  
    evenSem = new Semaphore(0);  
}
```

The argument passed to semaphore's constructor is the number of 'permits' available. For `oddSem` and `evenSem`, all `acquire()` calls will be blocked initially as they are initialized with 0. For `zeroSem`, the first `acquire()` call will succeed as it is initialized with 1. The code of the first method `PrintZero()` is as follows:

```
1. public void PrintZero() {  
2.     for (int i = 0; i < n; ++i) {  
3.         zeroSem.acquire();  
4.         System.out.print("0");  
5.         // release oddSem if i is even else release evenSem if i is odd  
6.         (i % 2 == 0 ? oddSem : evenSem).release();  
7.     }  
8. }
```

`PrintZero()` begins with a loop iterating from 0 till `n`(exclusive). The semaphore `zeroSem` is acquired and '0' is printed. A very significant line in this method is **line 6** in the loop. The modulus operator (%) gives the remainder of a division by the value following it. In our case, the current value is divided by 2 to determine if `i` is even or odd. If `i` is odd, then it means we just printed an odd number and the next number in the sequence will be an even number so, `evenSem` is released. In the same way if `i` is even then `oddSem` is released for printing the next odd number in the sequence. The second method `PrintOdd()` is shown below:

```
public void PrintOdd() {  
    for (int i = 1; i <= n; i += 2) {  
        oddSem.acquire();  
        System.out.print(i);  
        zeroSem.release();  
    }  
}
```

The loop iterates from 1 till `n` (inclusive) and `i` is incremented by 2 after each iteration to ensure that only odd numbers are printed. `oddSem` is acquired when `PrintZero()` releases it after determining that it is the turn

for an odd number to be printed. Since zero is required to be printed before every even or odd number, `zeroSem` is released after printing the odd number.

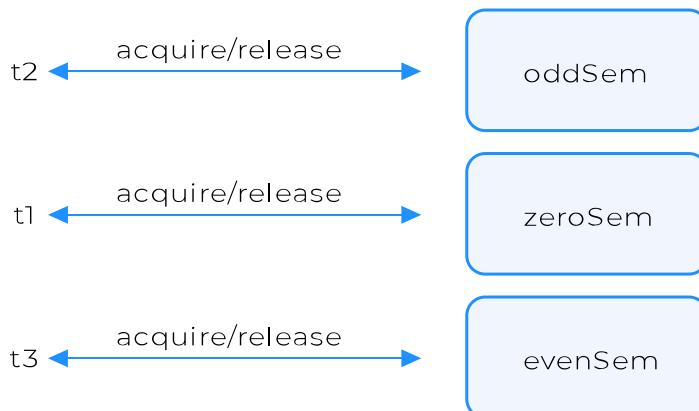
The last method of the class `PrintEven()` is shown below:

```
public void PrintEven() {
    for (int i = 2; i <= n; i += 2) {
        evenSem.acquire();
        System.out.print(i);
        zeroSem.release();
    }
}
```

`PrintEven()` operates in the same manner as `PrintOdd()` except that its loop begins from 2. `evenSem` is acquired if it is released by `PrintZero()` and an even number is printed. `zeroSem` is released for zero to be printed next. If `n` is reached, the loop breaks.

The working of this class can be seen as a lock-shift phenomenon where every method is given the control at its turn and blocked otherwise. `n` is manipulated by only one thread at a time.

Shared Resource



To test our solution, We will create 3 threads `t1`, `t2` and `t3` in `Main` class. `t1` prints 0, `t2` prints odd numbers and `t3` prints even numbers. The threads are started in random order.
import java.util.concurrent.*;

```
class PrintNumberSeries {
    private int n;
    private Semaphore zeroSem, oddSem, evenSem;

    public PrintNumberSeries(int n) {
        this.n = n;
```

```

zeroSem = new Semaphore(1);
oddSem = new Semaphore(0);
evenSem = new Semaphore(0);
}

public void PrintZero() {
    for (int i = 0; i < n; ++i) {
        try {
            zeroSem.acquire();
        }
        catch (Exception e) {
        }
        System.out.print("0");
        // release oddSem if i is even or else release evenSem if i is odd
        (i % 2 == 0 ? oddSem : evenSem).release();
    }
}

public void PrintEven() {
    for (int i = 2; i <= n; i += 2) {
        try {
            evenSem.acquire();
        }
        catch (Exception e) {
        }
        System.out.print(i);
        zeroSem.release();
    }
}

public void PrintOdd() {
    for (int i = 1; i <= n; i += 2) {
        try {
            oddSem.acquire();
        }
        catch (Exception e) {
        }
        System.out.print(i);
        zeroSem.release();
    }
}

class PrintNumberSeriesThread extends Thread {

    PrintNumberSeries zeo;
    String method;

    public PrintNumberSeriesThread(PrintNumberSeries zeo, String method){
        this.zeo = zeo;
    }
}

```

```

        this.method = method;
    }

public void run() {
    if ("zero".equals(method)) {
        try {
            zeo.PrintZero();
        }
        catch (Exception e) {
        }
    }
    else if ("even".equals(method)) {
        try {
            zeo.PrintEven();
        }
        catch (Exception e) {
        }
    }
    else if ("odd".equals(method)) {
        try {
            zeo.PrintOdd();
        }
        catch (Exception e) {
        }
    }
}

}

public class Main {

    public static void main(String[] args) {
        PrintNumberSeries zeo = new PrintNumberSeries(5);

        Thread t1 = new PrintNumberSeriesThread(zeo, "zero");
        Thread t2 = new PrintNumberSeriesThread(zeo, "even");
        Thread t3 = new PrintNumberSeriesThread(zeo, "odd");

        t2.start();
        t1.start();
        t3.start();

    }
}

```

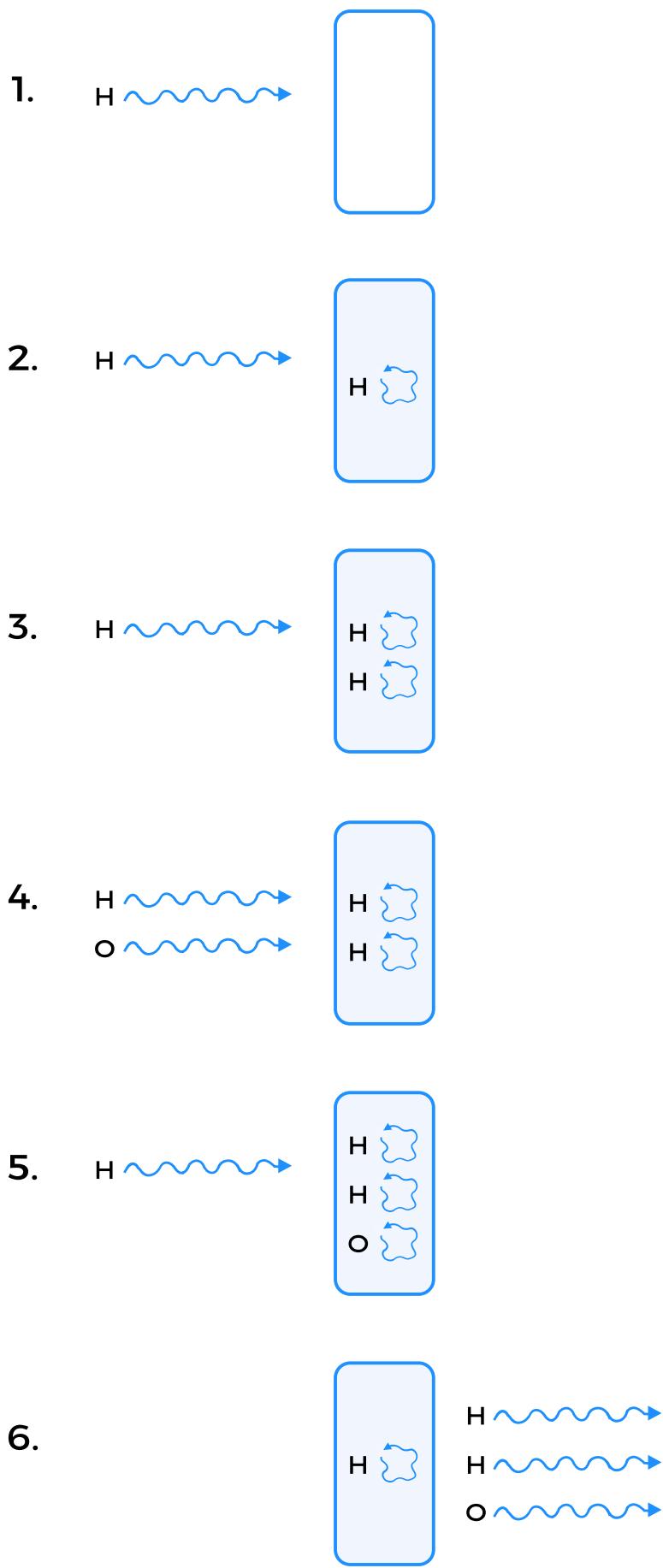
We were told to use three threads in the problem statement but the solution can be achieved using two threads as well. Since zero is printed before every number, we do not need to dedicate a special thread for it. We can simply print a zero before printing every odd or even number.

Build a Molecule

This problem simulated the creation of water molecule by grouping three threads representing Hydrogen and Oxygen atoms.

Problem Statement#

Suppose we have a machine that creates molecules by combining atoms. We are creating water molecules by joining one Oxygen and two Hydrogen atoms. The atoms are represented by threads. The machine will wait for the required atoms (threads), then group one Oxygen and two Hydrogen threads to simulate the creation of a molecule. The molecule then exists the machine. You have to ensure that one molecule is completed before moving onto the next molecule. If more than the required number of threads arrive, they will have to wait. The figure below explains the working of our machine:



Two Hydrogen threads are admitted in the machine as they arrive but when the third thread arrives in step 3, it is made to wait. When an Oxygen thread arrives in step 4, it is allowed to enter the machine. A water molecule is formed in step 5 which exists the machine in step 6. That is when the waiting Hydrogen thread is notified and the process of creating more molecules continues. The threads can arrive in any order which means that HHO, OHH and HOH are all valid outputs.

The code for the class is as follows:

```
lass H2OMachine {  
    public H2OMachine() {  
    }  
    public void HydrogenAtom() {  
    }  
    public void OxygenAtom() {  
    }  
}
```

The input to the machine can be in any order. Your program should enforce a 2:1 ratio for Hydrogen and Oxygen threads, and stop more than the required number of threads from entering the machine.

Solution#

Our molecule making machine is represented by the class `H2OMachine`, which contains two main methods; `HydrogenAtom()` and `OxygenAtom()`.

The problem is solved by using basic utility functions like `notify()` and `wait()`. The class consists of 3 private members: `sync` for synchronization, `molecule` which is a string array with a capacity of 3 elements (atoms) and `count` to store the current index of the molecule array.

```
class H2OMachine {  
    Object sync;  
    String[] molecule;  
    int count;  
  
    public H2OMachine() {  
    }  
  
    public void HydrogenAtom() {  
    }
```

```
    public void OxygenAtom() {  
    }  
}
```

The constructor initializes the `molecule` array with a capacity of 3 atoms and the integer `count` is initialized with 0.

```
public H2OMachine() {  
    molecule = new String[3];  
    count = 0;  
    sync = new Object();  
}
```

For synchronization purpose, the entire logic of `HydrogenAtom()` is wrapped in `sync`. First of all, we check the frequency of Hydrogen atom in the `molecule` array by using `frequency()` function found in the **Collections** library of Java. The function deals the array `molecule` as an `ArrayList` and checks the count of Hydrogen atoms in it. If the array has reached its capacity of 2 Hydrogen atoms, then the thread should wait for space in a new molecule. If the frequency of Hydrogen is less than 2 it means space is available in the current molecule. Hence, H is placed in the array and `count` is incremented. So far, the code of `HydrogenAtom()` is as follows:

```
public void HydrogenAtom() {  
    synchronized (sync) {  
  
        // if 2 hydrogen atoms already exist  
        while (Collections.frequency(Arrays.asList(molecule), "H") == 2) {  
            sync.wait();  
        }  
  
        molecule[count] = "H";  
        count++;  
    }  
}
```

In case `molecule` is full and `count` is 3, then print the `molecule` and exit the machine. The array `molecule` is reset (initialized with null) and `count` goes back to 0 for a new molecule to be built. At the end of the method, the waiting threads (atoms) are notified using `notifyAll()`. The complete code for `HydrogenAtom()` is given below:

```
public void HydrogenAtom() {  
    synchronized (sync) {  
  
        // if 2 hydrogen atoms already exist  
        while (Collections.frequency(Arrays.asList(molecule), "H") == 2) {  
            sync.wait();  
        }  
  
        molecule[count] = "H";  
        count++;  
    }  
}
```

```

molecule[count] = "H";
count++;

// if molecule is full, then exit.
if(count == 3) {
    for (String element: molecule) {
        System.out.print(element);
    }
    Arrays.fill(molecule,null);
    count = 0;
}
sync.notifyAll();
}
}

```

The second method `OxygenAtom()` is the same as `HydrogenAtom()` with the only difference of the atom frequency check in the array `molecule`. If it contains one Oxygen atom, then the calling thread goes into `wait()`. If the count of Oxygen atom is not equal to 1 in the `molecule`, then an Oxygen atom "O" is placed in the next available space. The complete code of `OxygenAtom()` is shown below:

```

public void oxygen() {
    synchronized (sync) {

        // if 1 oxygen atom already exists
        while (Collections.frequency(Arrays.asList(molecule),"O") == 1) {
            sync.wait();
        }

        molecule[count] = "O";
        count++;

        // if molecule is full, then exit.
        if(count == 3) {
            for (String element: molecule) {
                System.out.print(element);
            }
            Arrays.fill(molecule,null);
            count = 0;
        }
        sync.notifyAll();
    }
}

```

The complete code for the solution is as follows:

```

class H2OMachine {

    Object sync;
    String[] molecule;
    int count;

    public H2OMachine() {

```

```

molecule = new String[3];
count = 0;
sync = new Object();
}

public void HydrogenAtom() {
    synchronized (sync) {

        // if 2 hydrogen atoms already exist
        while (Collections.frequency(Arrays.asList(molecule), "H") == 2
    ) {
            sync.wait();
        }

        molecule[count] = "H";
        count++;

        // if molecule is complete, then exit.
        if(count == 3) {
            for (String element: molecule) {
                System.out.print(element);
            }
            Arrays.fill(molecule,null);
            count = 0;
        }
        sync.notifyAll();
    }
}

public void OxygenAtom() throws InterruptedException {
    synchronized (sync) {

        // if 1 oxygen atom already exists
        while (Collections.frequency(Arrays.asList(molecule), "O") == 1
    ) {
            sync.wait();
        }

        molecule[count] = "O";
        count++;

        // if molecule is complete, then exit.
        if(count == 3) {
            for (String element: molecule) {
                System.out.print(element);
            }
            Arrays.fill(molecule,null);
            count = 0;
        }
        sync.notifyAll();
    }
}
}

```

We will be creating another class `H2OMachineThread` for multi-threading purpose. It takes an object of `H2OMachine` and calls the relevant method from the string passed to it.

```

class H2OMachineThread extends Thread {

    H2OMachine molecule;
    String atom;

    public H2OMachineThread(H2OMachine molecule, String atom){
        this.molecule = molecule;
        this.atom = atom;
    }

    public void run() {
        if ("H".equals(atom)) {
            try {
                molecule.HydrogenAtom();
            }
            catch (Exception e) {
            }
        }
        else if ("O".equals(atom)) {
            try {
                molecule.OxygenAtom();
            }
            catch (Exception e) {
            }
        }
    }
}

```

We will now be creating 4 threads in order to test our proposed solution. Same object of **H2OMachine** is passed to the 4 threads: **t1**, **t2**, **t3** and **t4**.

t1 and **t3** act as Hydrogen atoms trying to enter the machine where as **t2** and **t4** act as Oxygen atoms. It can be seen from the output that only 1 molecule of H₂O exits the machine while the extra Oxygen atom is not utilized.

```

import java.util.Arrays;
import java.util.Collections;

class H2OMachine {

    Object sync;
    String[] molecule;
    int count;

    public H2OMachine() {
        molecule = new String[3];
        count = 0;
        sync = new Object();
    }

    public void HydrogenAtom() {
        synchronized (sync) {

```

```

// if 2 hydrogen atoms already exist
while (Collections.frequency(Arrays.asList(molecule),"H") == 2) {
    try {
        sync.wait();
    }
    catch (Exception e) {
    }
}

molecule[count] = "H";
count++;

// if molecule is complete, then exit.
if(count == 3) {
    for (String element: molecule) {
        System.out.print(element);
    }
    Arrays.fill(molecule,null);
    count = 0;
}
sync.notifyAll();
}

public void OxygenAtom() throws InterruptedException {
    synchronized (sync) {

        // if 1 oxygen atom already exists
        while (Collections.frequency(Arrays.asList(molecule),"O") == 1) {
            try {
                sync.wait();
            }
            catch (Exception e) {
            }
        }

        molecule[count] = "O";
        count++;

        // if molecule is complete, then exit.
        if(count == 3) {
            for (String element: molecule) {
                System.out.print(element);
            }
            Arrays.fill(molecule,null);
            count = 0;
        }
        sync.notifyAll();
    }
}

```

```

        }
    }
class H2OMachineThread extends Thread {

    H2OMachine molecule;
    String atom;

    public H2OMachineThread(H2OMachine molecule, String atom){
        this.molecule = molecule;
        this.atom = atom;
    }

    public void run() {
        if ("H".equals(atom)) {
            try {
                molecule.HydrogenAtom();
            }
            catch (Exception e) {
            }
        }
        else if ("O".equals(atom)) {
            try {
                molecule.OxygenAtom();
            }
            catch (Exception e) {
            }
        }
    }
}

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

        H2OMachine molecule = new H2OMachine();

        Thread t1 = new H2OMachineThread(molecule,"H");
        Thread t2 = new H2OMachineThread(molecule,"O");
        Thread t3 = new H2OMachineThread(molecule,"H");
        Thread t4 = new H2OMachineThread(molecule,"O");

        t2.start();
        t1.start();
        t4.start();
        t3.start();
    }
}

```

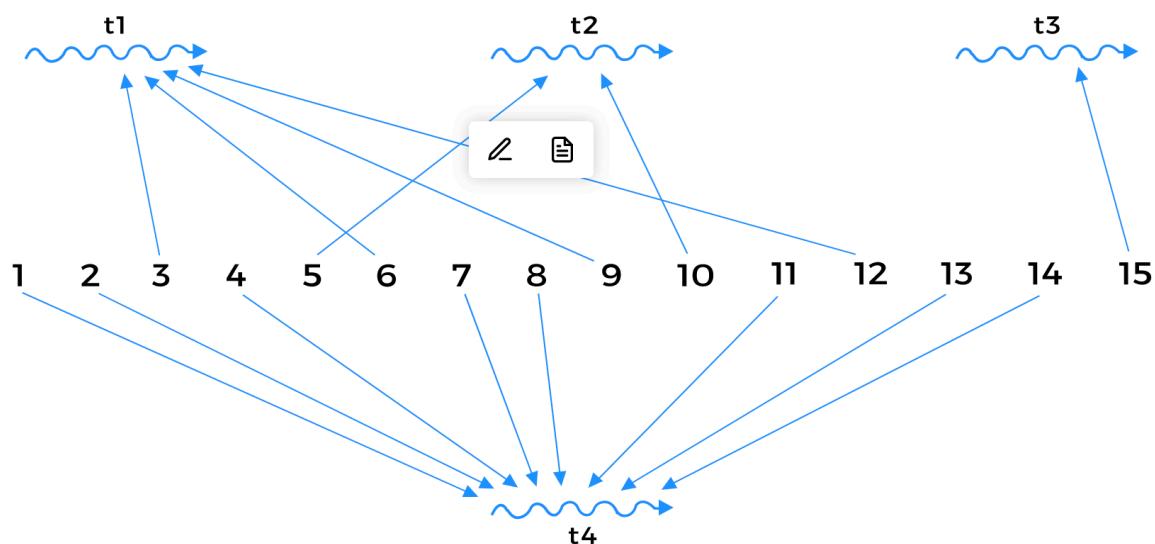
Fizz Buzz Problem

This problem explores a multi-threaded solution to the very common Fizz Buzz programming task

Problem Statement#

FizzBuzz is a common interview problem in which a program prints a number series from 1 to n such that for every number that is a multiple of 3 it prints "fizz", for every number that is a multiple of 5 it prints "buzz" and for every number that is a multiple of both 3 and 5 it prints "fizzbuzz". We will be creating a multi-threaded solution for this problem.

Suppose we have four threads t_1 , t_2 , t_3 and t_4 . Thread t_1 checks if the number is divisible by 3 and prints **fizz**. Thread t_2 checks if the number is divisible by 5 and prints **buzz**. Thread t_3 checks if the number is divisible by both 3 and 5 and prints **fizzbuzz**. Thread t_4 prints numbers that are not divisible by 3 or 5. The workflow of the program is shown below:



The code for the class is as follows:

```
class MultithreadedFizzBuzz {  
    private int n;  
    public MultithreadedFizzBuzz(int n) {  
        this.n = n;  
    }  
    public void fizz() {  
        System.out.print("fizz");  
    }  
    public void buzz() {  
        System.out.print("buzz");  
    }  
}
```

```

    }

    public void fizzbuzz() {
        System.out.print("fizzbuzz");
    }

    public void number(int num) {
        System.out.print(num);
    }
}

```

For an input integer n , the program should output a string containing the words fizz, buzz and fizzbuzz representing certain numbers. For example, for $n = 15$, the output should be: 1, 2, fizz, 4, buzz, fizz, 7, 8, fizz, buzz, 11, fizz, 13, 14, fizzbuzz.

Solution#

We will solve this problem using the basic Java functions; `wait()` and `notifyAll()`. The basic structure of the class is given below.

```

class MultithreadedFizzBuzz {

    private int n;
    private int num = 1;

    public MultithreadedFizzBuzz(int n) {
    }
    public void fizzbuzz() {
    }
    public void fizz() {
    }
    public void buzz() {
    }
    public void number() {
    }
}

```

The `MultithreadedFizzBuzz` class contains 2 private members: `n` and `num`.

`n` is the last number of the series to be printed whereas `num` represents the current number to be printed. It is initialized with 1.

The constructor `MultithreadedFizzBuzz()` initializes `n` with the input taken from the user.

```

public MultithreadedFizzBuzz(int n) {
    this.n = n;
}

```

The second function in the class, `fizz()` prints "fizz" only if the current number is divisible by 3. The first loop checks if `num` (current number) is smaller than or equal to `n`(user input). Then `num` is checked for its divisibility by 3. We check if `num` is divisible by 3 and not by 5 because some multiples of 3 are also multiples of 5. If the condition is met, then "fizz" is printed and `num` is incremented. The waiting threads are notified via `notifyAll()`. If the condition is not met, the thread goes into `wait()`.

```
public synchronized void fizz() throws InterruptedException {
    while (num <= n) {
        if (num % 3 == 0 && num % 5 != 0) {
            System.out.println("fizz");
            num++;
            notifyAll();
        }
        else {
            wait();
        }
    }
}
```

The next function `buzz()` works in the same manner as `fizz()`. The only difference here is the check to see if `num` is divisible by 5 and not by 3. The reasoning is the same: some multiples of 5 are also multiples of 3 and those numbers should not be printed by this function. If the condition is met, then "buzz" is printed otherwise the thread will `wait()`.

```
public synchronized void buzz() throws InterruptedException {
    while (num <= n) {
        if (num % 3 != 0 && num % 5 == 0) {
            System.out.println("buzz");
            num++;
            notifyAll();
        }
        else {
            wait();
        }
    }
}
```

The next function `fizzbuzz()` prints "fizzbuzz" if the current number in the series is divisible by both 3 and 5. A multiple of 15 is divisible by 3 and 5 both so `num` is checked for its divisibility by 15. After printing "fizzbuzz", `num` is incremented and waiting threads are notified via `notifyAll()`. If `num` is not divisible by 15 then the thread goes into `wait()`.

```
public synchronized void fizzbuzz() throws InterruptedException {
    while (num <= n) {
        if (num % 15 == 0) {
```

```

        System.out.println("fizzbuzz");
        num++;
        notifyAll();
    }
    else {
        wait();
    }
}
}

```

The last function `number()` checks if `num` is neither divisible by 3 nor by 5, then prints the `num`.

```

public synchronized void number() throws InterruptedException {
    while (num <= n) {
        if (num % 3 != 0 && num % 5 != 0) {
            System.out.println(num);
            num++;
            notifyAll();
        }
        else {
            wait();
        }
    }
}

```

The complete code for `MultithreadedFizzBuzz` is as follows

```

class MultithreadedFizzBuzz {

    private int n;
    private int num = 1;

    public MultithreadedFizzBuzz(int n) {
        this.n = n;
    }

    public synchronized void fizz() throws InterruptedException {
        while (num <= n) {
            if (num % 3 == 0 && num % 5 != 0) {
                System.out.println("fizz");
                num++;
                notifyAll();
            } else {
                wait();
            }
        }
    }

    public synchronized void buzz() throws InterruptedException {
        while (num <= n) {
            if (num % 3 != 0 && num % 5 == 0) {
                System.out.println("buzz");
                num++;
                notifyAll();
            } else {

```

```

        wait();
    }
}

public synchronized void fizzbuzz() throws InterruptedException {
    while (num <= n) {
        if (num % 15 == 0) {
            System.out.println("fizzbuzz");
            num++;
            notifyAll();
        } else {
            wait();
        }
    }
}

public synchronized void number() throws InterruptedException {
    while (num <= n) {
        if (num % 3 != 0 && num % 5 != 0) {
            System.out.println(num);
            num++;
            notifyAll();
        } else {
            wait();
        }
    }
}
}

```

We will be creating another class **FizzBuzzThread** for multithreading purpose. It takes an object of **MultithreadedFizzBuzz** and calls the relevant method from the string passed to it.

```

class FizzBuzzThread extends Thread {

    MultithreadedFizzBuzz obj;
    String method;

    public FizzBuzzThread(MultithreadedFizzBuzz obj, String method){
        this.obj = obj;
        this.method = method;
    }

    public void run() {
        if ("Fizz".equals(method)) {
            try {
                obj.fizz();
            }
            catch (Exception e) {
            }
        }
        else if ("Buzz".equals(method)) {
            try {
                obj.buzz();
            }
            catch (Exception e) {
            }
        }
    }
}

```

```
        }
    }
    else if ("FizzBuzz".equals(method)) {
        try {
            obj.fizzbuzz();
        }
        catch (Exception e) {
        }
    }
    else if ("Number".equals(method)) {
        try {
            obj.number();
        }
        catch (Exception e) {
        }
    }
}
```

To test our solution, we will be making 4 threads: **t1**, **t2**, **t3** and **t4**. Three threads will check for divisibility by 3, 5 and 15 and print **fizz**, **buzz**, and **fizzbuzz** accordingly. Thread **t4** prints numbers that are not divisible by 3 or 5.

```
class MultithreadedFizzBuzz {  
    private int n;  
    private int num = 1;  
  
    public MultithreadedFizzBuzz(int n) {  
        this.n = n;  
    }  
    public synchronized void fizz() throws InterruptedException {  
        while (num <= n) {  
            if (num % 3 == 0 && num % 5 != 0) {  
                System.out.println("Fizz");  
                num++;  
                notifyAll();  
            } else {  
                wait();  
            }  
        }  
    }  
  
    public synchronized void buzz() throws InterruptedException {  
        while (num <= n) {  
            if (num % 3 != 0 && num % 5 == 0) {  
                System.out.println("Buzz");  
                num++;  
                notifyAll();  
            } else {  
                wait();  
            }  
        }  
    }  
}
```

```

        }
    }
}

public synchronized void fizzbuzz() throws InterruptedException {
    while (num <= n) {
        if (num % 15 == 0) {
            System.out.println("FizzBuzz");
            num++;
            notifyAll();
        } else {
            wait();
        }
    }
}

public synchronized void number() throws InterruptedException {
    while (num <= n) {
        if (num % 3 != 0 && num % 5 != 0) {
            System.out.println(num);
            num++;
            notifyAll();
        } else {
            wait();
        }
    }
}

class FizzBuzzThread extends Thread {

    MultithreadedFizzBuzz obj;
    String method;

    public FizzBuzzThread(MultithreadedFizzBuzz obj, String method){
        this.obj = obj;
        this.method = method;
    }

    public void run() {
        if ("Fizz".equals(method)) {
            try {
                obj.fizz();
            }
            catch (Exception e) {
            }
        }
        else if ("Buzz".equals(method)) {
            try {
                obj.buzz();
            }
            catch (Exception e) {
            }
        }
    }
}

```

```

        }
        catch (Exception e) {
        }
    }
    else if ("FizzBuzz".equals(method)) {
        try {
            obj.fizzbuzz();
        }
        catch (Exception e) {
        }
    }
    else if ("Number".equals(method)) {
        try {
            obj.number();
        }
        catch (Exception e) {
        }
    }
}

class main
{
    public static void main(String[] args) {
        MultithreadedFizzBuzz obj = new MultithreadedFizzBuzz(15);

        Thread t1 = new FizzBuzzThread(obj,"Fizz");
        Thread t2 = new FizzBuzzThread(obj,"Buzz");
        Thread t3 = new FizzBuzzThread(obj,"FizzBuzz");
        Thread t4 = new FizzBuzzThread(obj,"Number");

        t2.start();
        t1.start();
        t4.start();
        t3.start();
    }
}

```

Java Concurrency Reference

Setting-up Threads

This lesson discusses how threads can be created in Java.

Creating Threads#

To use threads, we need to first create them. In the Java language framework, there are multiple ways of setting up threads.

Runnable Interface#

When we create a thread, we need to provide the created thread code to execute or in other words we need to tell the thread what *task* to execute. The code can be provided as an object of a class that implements the `Runnable` interface. As the name implies, the interface forces the implementing class to provide a `run` method which in turn is invoked by the thread when it starts.

The runnable interface is the basic abstraction to represent a logical task in Java.

```
class Demonstration {  
    public static void main( String args[] ) {  
        Thread t = new Thread(new Runnable() {  
  
            public void run() {  
                System.out.println("Say Hello");  
            }  
        });  
        t.start();  
    }  
}
```

We defined an anonymous class inside the `Thread` class's constructor and an instance of it is instantiated and passed into the `Thread` object. Personally, I feel anonymous classes decrease readability and would prefer to create a separate class implementing the `Runnable` interface. An instance of the implementing class can then be passed into the `Thread` object's constructor. Let's see how that could have been done.

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

```

ExecuteMe executeMe = new ExecuteMe();
Thread t = new Thread(executeMe);
t.start();
}
}

class ExecuteMe implements Runnable {

public void run() {
    System.out.println("Say Hello");
}

}

```

Subclassing Thread class#

The second way to set-up threads is to subclass the `Thread` class itself as shown below.

```

class Demonstration {
    public static void main( String args[] ) throws Exception {
        ExecuteMe executeMe = new ExecuteMe();
        executeMe.start();
        executeMe.join();

    }
}

class ExecuteMe extends Thread {

    @Override
    public void run() {
        System.out.println("I ran after extending Thread class");
    }

}

```

The con of the second approach is that one is forced to extend the `Thread` class which limits code's flexibility. Passing in an object of a class implementing the `Runnable` interface may be a better choice in most cases.

In next lesson, we'll study ways of manipulating threads

Basic Thread Handling

This lesson shows various thread handling methods with examples

Joining Threads#

In the previous section we discussed how threads can be created. The astute reader would realize that a thread is always created by another thread except for the main application thread. Study the following code snippet. The innerThread is created by the thread which executes the `main` method. You may wonder what happens to the innerThread if the main thread finishes execution before the innerThread is done?

```
class Demonstration {  
    public static void main( String args[] ) throws InterruptedException {  
  
        ExecuteMe executeMe = new ExecuteMe();  
        Thread innerThread = new Thread(executeMe);  
        innerThread.setDaemon(true);  
        innerThread.start();  
    }  
}  
  
class ExecuteMe implements Runnable {  
  
    public void run() {  
        while (true) {  
            System.out.println("Say Hello over and over again.");  
            try {  
                Thread.sleep(500);  
            } catch (InterruptedException ie) {  
                // swallow interrupted exception  
            }  
        }  
    }  
}
```

If you execute the above code, you'll see no output. That is because the main thread exits right after starting the innerThread. Once it exits, the JVM also kills the spawned thread. On **line 6** we mark the innerThread thread as a *daemon* thread, which we'll talk about shortly, and is responsible for innerThread being killed as soon as the main thread completes execution. Do bear in mind, that if the main thread context switches just after executing **line 7**, we may see some ouput form the innerThread, till the main thread is context switched back in and exits.

If we want the main thread to wait for the innerThread to finish before proceeding forward, we can direct the main thread to suspend its execution by calling `join` method on the innerThread object right after we `start` the innerThread. The change would look like the following.

```
Thread innerThread = new Thread(executeMe);
innerThread.start();
innerThread.join();
```

If we didn't execute `join` on innerThread and let the main thread continue after innerThread was spawned then the innerThread may get killed by the JVM upon main thread's completion.

Daemon Threads#

A daemon thread runs in the background but as soon as the main application thread exits, all daemon threads are killed by the JVM. A thread can be marked daemon as follows:

```
innerThread.setDaemon(true);
```

Note that in case a spawned thread isn't marked as daemon then even if the main thread finishes execution, JVM will wait for the spawned thread to finish before tearing down the process.

Sleeping Threads#

A thread can be made dormant for a specified period using the `sleep` method. However, be wary to not use sleep as a means for coordination among threads. It is a common newbie mistake. Java language framework offers other constructs for thread synchronization that'll be discussed later.

```
class SleepThreadExample {
    public static void main( String args[] ) throws Exception {
        ExecuteMe executeMe = new ExecuteMe();
        Thread innerThread = new Thread(executeMe);
        innerThread.start();
        innerThread.join();
        System.out.println("Main thread exiting.");
    }
    static class ExecuteMe implements Runnable {

        public void run() {
            System.out.println("Hello. innerThread going to sleep");
        }
    }
}
```

```

        try {
            Thread.sleep(1000);
        } catch (InterruptedException ie) {
            // swallow interrupted exception
        }
    }
}
}

```

In the above example, the innerThread is made to sleep for 1 second and from the output of the program, one can see that main thread exits only after innerThread is done processing. If we remove the `join` statement on **line-6**, then the main thread may print its statement before innerThread is done executing.

Interrupting Threads#

In the previous code snippets, we wrapped the calls to `join` and `sleep` in try/catch blocks. Imagine a situation where if a rogue thread sleeps forever or goes into an infinite loop, it can prevent the spawning thread from moving ahead because of the `join` call. Java allows us to force such a misbehaved thread to come to its senses by interrupting it. An example appears below.

```

class HelloWorld {
    public static void main( String args[] ) throws InterruptedException {
        ExecuteMe executeMe = new ExecuteMe();
        Thread innerThread = new Thread(executeMe);
        innerThread.start();

        // Interrupt innerThread after waiting for 5 seconds
        System.out.println("Main thread sleeping at " + System.currentTimeMillis() / 1000);
        Thread.sleep(5000);
        innerThread.interrupt();
        System.out.println("Main thread exiting at " + System.currentTimeMillis() / 1000);
    }

    static class ExecuteMe implements Runnable {

        public void run() {
            try {
                // sleep for a thousand minutes
                System.out.println("innerThread goes to sleep at " + System.currentTimeMillis() /
1000);
                Thread.sleep(1000 * 1000);
            } catch (InterruptedException ie) {

```

```
        System.out.println("innerThread interrupted at " + System.currentTimeMillis() /  
1000);  
    }  
}  
}  
}  
}
```

Executor Framework

This lesson discusses thread management using executors.

Creating and running individual threads for small applications is acceptable however if you are writing an enterprise-grade application with several dozen threads then you'll likely need to offload thread management in your application to library classes which free a developer from worrying about thread house-keeping.

Task#

A task is a logical unit of work. Usually, a task should be independent of other tasks so that it can be completed by a single thread. A task can be represented by an object of a class implementing the `Runnable` interface. We can consider HTTP requests being fielded by a web-server as tasks that need to be processed. A database server handling client queries can similarly be thought of as independent tasks.

Executor Framework#

In Java, the primary abstraction for executing logical tasks units is the Executor framework and not the Thread class. The classes in the Executor framework separate out:

- Task Submission
- Task Execution

The framework allows us to specify different policies for task execution. Java offers three interfaces, which classes can implement to manage thread lifecycle. These are:

- `Executor Interface`
- `ExecutorService`
- `ScheduledExecutorService`

The `Executor` interface forms the basis for the asynchronous task execution framework in Java.

You don't need to create your own executor class as Java's `java.util.concurrent` package offers several types of executors that are suitable for different scenarios. However, as an example, we create a dumb executor which implements the Executor Interface.

```
import java.util.concurrent.Executor;
class ThreadExecutorExample {

    public static void main( String args[] ) {
        DumbExecutor myExecutor = new DumbExecutor();
        MyTask myTask = new MyTask();
        myExecutor.execute(myTask);
    }

    static class DumbExecutor implements Executor {
        // Takes in a runnable interface object
        public void execute(Runnable runnable) {
            Thread newThread = new Thread(runnable);
            newThread.start();
        }
    }

    static class MyTask implements Runnable {
        public void run() {
            System.out.println("Mytask is running now ...");
        }
    }

}
```

The `Executor` requires implementing classes to define a method `execute(Runnable runnable)` which takes in an object of interface `Runnable`. Fortunately, we don't need to define complex executors as Java already provides several that we'll explore in following chapters.

Executor Implementations

Executors are based on consumer-producer patterns. The tasks we produce for processing are consumed by threads. To better our understanding of how threads behave, imagine you are hired by a hedge fund on Wall Street and you are asked to design a method that can process client purchase orders as soon as possible. Let's see what are the possible ways to design this method.

Sequential Approach#

The method simply accepts an order and tries to execute it. The method blocks other requests till it has completed processing the current request.

```
void receiveAndExecuteClientOrders() {  
    while (true) {  
        Order order = waitForNextOrder();  
        order.execute();  
    }  
}
```

You'll write the above code if you have never worked with concurrency. It sequentially processes each buy order and will not be *responsive* or have acceptable *throughput*.

Unbounded Thread Approach#

A newbie would fix the code above like so:

```
void receiveAndExecuteClientOrdersBetter() {  
    while (true) {  
        final Order order = waitForNextOrder();  
  
        Thread thread = new Thread(new Runnable() {  
  
            public void run() {  
                order.execute();  
            }  
        });  
  
        thread.start();  
    }  
}
```

The above approach is an improvement over the sequential approach. The program now accepts an order and spawns off a thread to handle the order execution. The problem, however, is that now the application spawns off an unlimited number of threads. Creating threads without bound is not a wise approach for the following reasons:

- Thread creation and teardown isn't for free.
- Active threads consume memory even if they are idle. If there are less number of processors than threads then several of them will sit idle tying up memory.

- There is usually a limit imposed by JVM and the underlying OS on the number of threads that can be created.

Note that the above improvement may still make the application unresponsive. Imagine if several hundred requests are received between the time it takes for the method to receive an order request and spawn off a thread to deal with the request. In such a scenario, the method will end up with a growing backlog of requests and may cause the program to crash.

The next lesson introduces Threadpools which mitigate several of the issues we discussed here.

Thread Pools

This lesson introduces thread pools and their utility in concurrent programming.

Thread Pools#

Thread pools in Java are implementations of the `Executor` interface or any of its sub-interfaces. Thread pools allow us to decouple task submission and execution. We have the option of exposing an executor's configuration while deploying an application or switching one executor for another seamlessly.

A thread pool consists of homogenous worker threads that are assigned to execute tasks. Once a worker thread finishes a task, it is returned to the pool. Usually, thread pools are bound to a queue from which tasks are dequeued for execution by worker threads.

A thread pool can be tuned for the size of the threads it holds. A thread pool may also replace a thread if it dies of an unexpected exception. Using a thread pool immediately alleviates from the ills of manual creation of threads.

- There's no latency when a request is received and processed by a thread because no time is lost in creating a thread.
- The system will not go out of memory because threads are not created without any limits
- Fine tuning the thread pool will allow us to control the throughput of the system. We can have enough threads to keep all processors busy but not so many as to overwhelm the system.
- The application will degrade gracefully if the system is under load.

Below is the updated version of the stock order method using a thread pool.

```
void receiveAndExecuteClientOrdersBest() {  
    int expectedConcurrentOrders = 100;  
    Executor executor = Executors.newFixedThreadPool(expectedConcurrentOrders);  
  
    while (true) {  
        final Order order = waitForNextOrder();  
  
        executor.execute(new Runnable() {  
            public void run() {  
                order.execute();  
            }  
        });  
    }  
}
```

In the above code we have used the factory method exposed by the `Executors` class to get an instance of a thread pool. We discuss the different type of thread pools available in Java in the next section.

Types of Thread Pools

This lesson details the different types of thread pools available in the Java class library.

Java has preconfigured thread pool implementations that can be instantiated using the factory methods of the `Executors` class. The important ones are listed below:

- **newFixedThreadPool:** This type of pool has a fixed number of threads and any number of tasks can be submitted for execution. Once a thread finishes a task, it can be reused to execute another task from the queue.
- **newSingleThreadExecutor:** This executor uses a single worker thread to take tasks off of queue and execute them. If the thread dies unexpectedly, then the executor will replace it with a new one.
- **newCachedThreadPool:** This pool will create new threads as required and use older ones when they become available. However, it'll terminate threads that remain idle for a certain configurable period of time to conserve memory. This pool can be a good choice for short-lived asynchronous tasks.
- **newScheduledThreadPool:** This pool can be used to execute tasks periodically or after a delay.

There is also another kind of pool which we'll only mention in passing as it's not widely used: `ForkJoinPool`. A prefconfigured version of it can be instantiated using the factory method `Executors.newWorkStealingPool()`. These pools are used for tasks which *fork* into smaller subtasks and then *join* results once the subtasks are finished to give an uber result. It's essentially the divide and conquer paradigm applied to tasks.

Using thread pools we are able to control the order in which a task is executed, the thread in which a task is executed, the maximum number of tasks that can be executed concurrently, maximum number of tasks that can be queued for execution, the selection criteria for rejecting tasks when the system is overloaded and finally actions to take before or after execution of tasks.

Executor Lifecycle#

An executor has the following stages in its lify-cycle:

- Running
- Shutting Down
- Terminated

As mentioned earlier, JVM can't exit unless all non-daemon thread have terminated. Executors can be made to shutdown either abruptly or gracefully. When doing the former, the executor attempts to cancel all tasks in progress and doesn't work on any enqueued ones, whereas when doing the latter, the executor gives a chance for tasks already in execution to complete but also completes the enqueued tasks. If shutdown is initiated then the executor will refuse to accept new tasks and if any are submitted, they can be handled by providing a `RejectedExecutionHandler`.

An Example: Timer vs ScheduledThreadPool

As an example, we'll compare and contrast using a timer and a pool to schedule periodic or delayed threads.

Timer#

The Achilles' heel of the `Timer` class is its use of a single thread to execute submitted tasks. Timer has a single worker thread that attempts to execute all user submitted tasks. Issues with this approach are detailed below:

- If a task misbehaves and never terminates, all other tasks would not be executed
- If a task takes too long to execute, it can block timely execution of other tasks. Say two tasks are submitted and the first is scheduled to execute after 100ms and the second is scheduled to execute after 500ms. Now if the first task takes 5 minutes to execute then the second task would get delayed by 5 minutes rather than the intended 500ms.
- In the above example, if the second task is scheduled to run periodically after every 500ms, then when it finally gets a chance to run after 5 minutes, it'll run for all the times it missed its turns, one after the other, without any delay between consecutive runs.

```
import java.util.Timer;
import java.util.TimerTask;

class Demonstration {
    public static void main( String args[] ) throws Exception {
        Timer timer = new Timer();
        TimerTask badTask = new TimerTask() {

            @Override
            public void run() {

                // run forever
                while (true)
                    ;

            }
        };

        TimerTask goodTask = new TimerTask() {

            @Override
            public void run() {

                System.out.println("Hello I am a well-behaved task");
            }
        };
    }
}
```

```

        }
    };

    timer.schedule(badTask, 100);
    timer.schedule(goodTask, 500);

    // By three seconds, both tasks are expected to have launched
    Thread.sleep(3000);
}
}

Bad Use of Timer

```

Below is another example of `Timer`'s shortcoming. We schedule a task which throws a runtime exception and ends up killing the lone worker thread `Timer` possess. The subsequent submission of a task reports the *timer is canceled* when in fact the previously submitted task crashed the `Timer`.

```

import java.util.Timer;
import java.util.TimerTask;

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

        Timer timer = new Timer();
        TimerTask badTask = new TimerTask() {

            @Override
            public void run() {
                throw new RuntimeException("Something Bad Happened");
            }
        };

        TimerTask goodTask = new TimerTask() {

            @Override
            public void run() {
                System.out.println("Hello I am a well-behaved task");
            }
        };

        timer.schedule(badTask, 10);
        Thread.sleep(500);
        timer.schedule(goodTask, 10);
    }
}

```

ThreadPoolExecutor

Go through the most comprehensive and complete guide to working with the ThreadPoolExecutor class.

We'll cover the following

- Overview
- Example
- corePoolSize and maximumPoolSize
 - Setting corePoolSize equal to maximumPoolSize
 - Setting maximumPoolSize to an arbitrary high value
- Keep-alive
- ThreadFactory
- Queuing
 - Queuing Strategies
 - Direct handoffs
 - Unbounded queues
 - Bounded queues
 - Queue Manipulation
- Task Rejection
 - ThreadPoolExecutor.AbortPolicy
 - ThreadPoolExecutor.CallerRunsPolicy
 - ThreadPoolExecutor.DiscardPolicy
 - ThreadPoolExecutor.DiscardOldestPolicy
- Shutting down
- Hooks
- Conclusion

Overview#

In general, a thread pool is a group of threads instantiated and kept alive to execute submitted tasks. Thread pools can achieve better performance and throughput than creating an individual thread per task by circumventing the overhead associated with thread creation and destruction. Additionally, system resources can be better managed using a thread pool, which allows us to limit the number of threads in the system.

Generally the use of the `ThreadPoolExecutor` class is discouraged in the favor of thread pools that can be instantiated using the `Executors` factory methods. These thread pools come with pre-configured settings that are commonly used in most scenarios, however, the `ThreadPoolExecutor` comes with several knobs and parameters that can be fine-tuned to suit unusual use-cases. Before we delve into the `ThreadPoolExecutor` class, we'll list some of the thread pools provided by the `Executors` factory methods:

- `Executors.newCachedThreadPool()` - (unbounded thread pool, with automatic thread reclamation)
- `Executors.newFixedThreadPool(int)` (fixed size thread pool)
- `Executors.newSingleThreadExecutor()` (single background thread)
- `Executors.newScheduledThreadPool(int)` (fixed size thread pool supporting delayed and periodic task execution.)

Example#

Let's consider the constructor that takes-in the most arguments to instantiate the `ThreadPoolExecutor` class:

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          RejectedExecutionHandler handler)
```

We'll shortly study how each of these arguments impact the behavior of the executor. First, we'll start with a simple example program that demonstrates the use of the `ThreadPoolExecutor` class:

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

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

        // create an instance of the ThreadPoolExecutor
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(1, 5, 1,
```

```

        TimeUnit.MINUTES, new LinkedBlockingDeque<>(3), new
        ThreadPoolExecutor.AbortPolicy());

    try {
        // submit six tasks
        for (int i = 0; i < 6; i++) {
            threadPoolExecutor.submit(new Runnable() {
                @Override
                public void run() {
                    System.out.println("This is worker thread " + Thread.currentThread().getName() +
" executing");
                    try {
                        // simulate work by sleeping for 1 second
                        Thread.sleep(1000);
                    } catch (InterruptedException ie) {
                        // ignore for now
                    }
                }
            });
        }
    } finally {
        threadPoolExecutor.shutdown();
    }
}
}

```

corePoolSize and maximumPoolSize

#

The arguments `corePoolSize` and the `maximumPoolSize` together determine the number of threads that get created in the pool. The workflow is as follows:

- When the pool has less than `corePoolSize` threads and a new task arrives, a new thread is instantiated even if other threads in the pool are idle.
- When the pool has more than `corePoolSize` threads but less than `maximumPoolSize` threads then a new thread is only created if the queue that holds the submitted tasks is full.
- The maximum number of threads that can be created is capped by `maximumPoolSize`.

Both `corePoolSize` and `maximumPoolSize` can be changed dynamically after construction of the pool instance. Note that a newly instantiated pool creates core threads only when tasks start arriving in the queue. However,

this behavior can be tweaked by invoking one of the `prestartCoreThread()` or `prestartAllCoreThreads()` methods, which is a good idea when creating a pool with a non-empty queue.

Setting `corePoolSize` equal to `maximumPoolSize`#

If we set `corePoolSize` equal to `maximumPoolSize` we effectively create a fixed size thread pool.

Setting `maximumPoolSize` to an arbitrary high value#

Setting `maximumPoolSize` to an unbounded value such as `Integer.MAX_VALUE` allows the pool to accommodate an arbitrary number of concurrent tasks.

Keep-alive#

A thread pool will eliminate threads in excess of `corePoolSize` after `keepAliveTime` has elapsed. The `unit` argument specifies the `TimeUnit` for the passed-in value of `keepAliveTime`, which can be milliseconds, minutes, hours etc.

ThreadFactory#

The pool creates new threads using a `ThreadFactory`. The user has the choice to pass-in a factory of her own choice or let the `ThreadPoolExecutor` class choose the default. Usually, you would pass-in a thread factory argument if you want to change the thread name, thread group, priority, daemon status etc.

Queuing#

The `ThreadPoolExecutor` takes in a `BlockingQueue` as a parameter in its constructor. The queue is used to hold tasks submitted to the executor. The queue works in tandem with the pool's thread size.

- If fewer than `corePoolSize` threads are running when a new task is submitted, the executor prefers adding a new thread rather than queuing the task. Remember:
- If `corePoolSize` or more threads are running, the executor prefers queuing the task than creating a new thread.

- If the queue is full and creating a new thread would exceed `maximumPoolSize` the submitted task is rejected by the executor. We'll shortly explain the various policies that govern task rejection.

Queuing Strategies#

The choice of the queue we pass-in determines the queuing strategy for the executor. The queuing strategies are:

Direct handoffs#

Direct handoff design involves an object running in one thread syncing up with an object running in another thread to hand off a piece of information, event or task. The `SynchronousQueue` class can be used for implementing the direct handoff strategy. The `SynchronousQueue` doesn't have an internal capacity (not even 1) and an item can only be inserted in the queue if another thread is simultaneously removing it. The widget below demonstrates that if we add an item to a `SynchronousQueue` without another thread removing the item, the thread making the insert simply blocks. The execution for the widget below times out as the main thread gets blocked.

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

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        SynchronousQueue<Integer> synchronousQueue = new SynchronousQueue<>();

        // The following statement blocks the main thread as there is no corresponding
        // thread to dequeue the item being placed in the synchronous queue.
        synchronousQueue.put(7);
    }
}
```

Given this behavior it is obvious that if the `ThreadPoolExecutor` is initialized with `SynchronousQueue`, each new task submitted to the `ThreadPoolExecutor` is handed off by the queue to one of the pool threads for execution, i.e. the queue doesn't hold any tasks. However, if tasks submitted exceed `maximumPoolSize` the queue doesn't hold any tasks and if no free threads are available then the submitted tasks are rejected. Consider the example program below, where we set the `maximumPoolSize` to 5 and then attempt to submit 50 tasks. Each task sleeps for 1 second so the entire pool is hogged after 5 tasks are submitted. The 6th task when submitted has the executor throw the `RejectedExecutionException` to indicate that the task can't be accepted for execution by the thread pool. The tasks can't be queued by the `SynchronousQueue` and if no free thread is available a new one must be created but if the number of threads has already reached the maximum allowed number then the task is rejected.

```

import java.util.concurrent.*;

class Demonstration {

    public static void main( String args[] ) throws InterruptedException {
        // create a ThreadPoolExecutor with a SynchronousQueue to implement the direct handoff
        strategy. The pool has
        // a maximum of 5 threads. Since we aren't passing-in the RejectionHandler, the default
        AbortPolicy will be used.
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(1, 5, 1,
            TimeUnit.MINUTES, new SynchronousQueue());
    }

    int i = 0;
    try {

        // Try to submit 50 tasks
        for (; i < 50; i++) {
            threadPoolExecutor.execute(new Runnable() {
                @Override
                public void run() {
                    try {
                        // simulate work by sleeping for 1 second
                        System.out.println("Thread " + Thread.currentThread().getName() + " at work.");
                        Thread.sleep(1000);
                    } catch (InterruptedException ie) {
                        // ignore for now
                    }
                }
            });
        }
    } catch (RejectedExecutionException ree) {
        // Let's see which task gets rejected
        System.out.println("Task " + (i + 1) + " rejected.");
    } finally {
        // don't forget to shutdown the executor
        threadPoolExecutor.shutdown();

        // wait for the executor to shutdown
        threadPoolExecutor.awaitTermination(1, TimeUnit.HOURS);
    }
}
}

```

Folks with background in CSP or ADA would find the `SynchronousQueue` similar to *rendezvous channels*. Using the direct handoff strategy requires that the maximum allowed threads for a thread pool should be unbounded e.g. `Integer.MAX_VALUE` to avoid tasks being

rejected. However, this setting entails that the number of threads in the system can grow to be very large if tasks are submitted at a rate faster than they can be processed at. Direct handoff policy is useful when handling sets of requests that might have internal dependencies as lockups are avoided.

Unbounded queues#

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 in contrast to the direct handoff approach in which the number of threads can grow indefinitely. Consider the program below that uses the `LinkedBlockingQueue` without a defined capacity and only 5 tasks the same as the `corePoolSize` execute at any time. The rest pile-up in the queue.

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

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        // create a ThreadPoolExecutor with a LinkedBlockingDeque to implement the unbounded
        queue strategy. The pool has
        // a maximum of 5 threads. Since we aren't passing-in the RejectionHandler, the default
        AbortPolicy will be used.
        // Note that the maximumPoolSize setting doesn't have any effect since only corePoolSize
        threads are ever created
        // because the queue has indefinite (theoretically) capacity.
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(2, 5, 1,
            TimeUnit.MINUTES, new LinkedBlockingDeque<>());

        int i = 0;
        try {

            // Try to submit 20 tasks
            for (; i < 20; i++) {
                threadPoolExecutor.execute(new Runnable() {
                    @Override
                    public void run() {
                        try {
                            // simulate work by sleeping for 1 second
                            System.out.println("Thread " + Thread.currentThread().getName() + " at work.");
                            Thread.sleep(1000);
                        } catch (InterruptedException ie) {
                            // ignore for now
                        }
                    }
                });
            }
        }
    }
}
```

```

        });
    }
} catch (RejectedExecutionException ree) {
    // Let's see which task gets rejected
    System.out.println("Task " + (i + 1) + " rejected.");
} finally {
    // don't forget to shutdown the executor
    threadPoolExecutor.shutdown();

    // wait for the executor to shutdown
    threadPoolExecutor.awaitTermination(1, TimeUnit.HOURS);
}
}
}
}

```

The output of the above program is interesting to observe. Note, that only two threads, which is also the `corePoolSize`, ever execute the submitted tasks. At a time, only two tasks are executed while the rest queue-up and the queue grows without any bounds.

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.

```

import java.util.concurrent.*;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        // create a ThreadPoolExecutor with a LinkedBlockingDeque to implement the unbounded
        queue strategy. The pool has
        // a maximum of 5 threads. Since we aren't passing-in the RejectionHandler, the default
        AbortPolicy will be used.
        // The queue has a defined capacity so the setting maximumPoolSize does take effect.
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(2, 5, 1,
            TimeUnit.MINUTES, new LinkedBlockingDeque<>(5));

        int i = 0;
        try {

            // Try to submit 20 tasks
            for (; i < 20; i++) {
                threadPoolExecutor.execute(new Runnable() {
                    @Override
                    public void run() {
                        try {
                            // simulate work by sleeping for 1 second

```

```

        System.out.println("Thread " + Thread.currentThread().getName() + " at work.");
        Thread.sleep(1000);
    } catch (InterruptedException ie) {
        // ignore for now
    }
}
});
}
} catch (RejectedExecutionException ree) {
    // Let's see which task gets rejected
    System.out.println("Task " + (i + 1) + " rejected.");
} finally {
    // don't forget to shutdown the executor
    threadPoolExecutor.shutdown();

    // wait for the executor to shutdown
    threadPoolExecutor.awaitTermination(1, TimeUnit.HOURS);
}
}
}
}

```

In the above program the 11th task gets rejected and the first 10 get executed. This makes sense because the thread pool has a maximum of 5 threads, all of which start working on the first five submitted tasks. Thereafter the next 5 threads get queued-up in the queue which has a maximum capacity of 5. When the 11th task gets submitted, all five tasks are busy executing the first 5 tasks and the queue is full, therefore the 11th task is rejected.

Bounded queues#

The astute reader can deduce from the previous discussion that a tradeoff between the maximum threads and queue sizes exists. Constraining one allows the other to grow unbounded. The middle of the spectrum is to define a queue with a certain capacity and also set a limit on the maximum number of threads. Using a bounded queue with a finite maximum pool size helps prevent resource exhaustion in the system. However, using large queue sizes and small pools minimizes CPU usage, OS resources, and context-switching overhead, but can lead to artificially low throughput. In systems where threads occasionally block for I/O, a system may be able to schedule time for more threads than you otherwise allow. Using smaller queues generally requires larger pool sizes, which keeps CPUs busier but may encounter unacceptable scheduling overhead, which also decreases throughput.

Thus choosing a queuing strategy will involve looking at the particular characteristics of your system and picking an option suitable for your use case.

Queue Manipulation#

We can access the queue using the `getQueue()` method, however, use of this method for purposes other than debugging and monitoring is discouraged. Two other methods, namely, `remove(Runnable)` and `purge()` are available to assist in storage reclamation when large numbers of queued tasks become cancelled.

Task Rejection#

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. Tasks can also be rejected when they are submitted to an executor that has already been shutdown. There are four different policies that can be supplied to the executor to determine the course of action when tasks can't be accepted any more. These policies are represented by four classes that extend the `RejectedExecutionHandler` class. The executor invokes the `rejectedExecution()` method of the supplied `RejectedExecutionHandler` when a task is intended for rejection. We discuss them below:

ThreadPoolExecutor.AbortPolicy#

The abort policy simply throws the runtime `RejectedExecutionException` when a task can't be accepted. The previous widget demonstrates the use of the `ThreadPoolExecutor.AbortPolicy` when tasks get rejected if they can't be accommodated by the thread pool.

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.

The previous program is reproduced with the change of the `RejectionHandler` to `ThreadPoolExecutor.CallerRunsPolicy`:

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

```

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        // create a ThreadPoolExecutor
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(1, 5, 1,
            TimeUnit.MINUTES, new LinkedBlockingDeque<>(5), new
            ThreadPoolExecutor.CallerRunsPolicy());

        int i = 0;
        try {

            // Try to submit 20 tasks
            for (; i < 20; i++) {
                threadPoolExecutor.execute(new Runnable() {
                    @Override
                    public void run() {
                        try {
                            // simulate work by sleeping for 1 second
                            System.out.println("Thread " + Thread.currentThread().getName() + " at work.");
                            Thread.sleep(1000);
                        } catch (InterruptedException ie) {
                            // ignore for now
                        }
                    }
                });
            }

            } catch (RejectedExecutionException ree) {
                // Let's see which task gets rejected
                System.out.println("Task " + (i + 1) + " rejected.");
            } finally {
                // don't forget to shutdown the executor
                threadPoolExecutor.shutdown();

                // wait for the executor to shutdown
                threadPoolExecutor.awaitTermination(1, TimeUnit.HOURS);
            }
        }
    }
}

```

Notice that in the output of the above program, when the thread pool can't accept any more tasks, the main thread that is submitting the tasks, is itself pulled-in to execute the submitted task. Consequently, the submission of new tasks slows down as the main thread now executes the task itself.

ThreadPoolExecutor.DiscardPolicy#

A task that can't be executed is simply dropped. In the program below, the rejection policy has been changed to `ThreadPoolExecutor.DiscardPolicy`:

```
import java.util.concurrent.*;  
  
class Demonstration {  
    public static void main( String args[] ) throws InterruptedException {  
        // create a ThreadPoolExecutor  
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(1, 5, 1,  
            TimeUnit.MINUTES, new LinkedBlockingDeque<>(5), new  
            ThreadPoolExecutor.DiscardPolicy());  
  
        int i = 0;  
        try {  
  
            // Try to submit 20 tasks  
            for (; i < 20; i++) {  
                threadPoolExecutor.execute(new Runnable() {  
                    @Override  
                    public void run() {  
                        try {  
                            // simulate work by sleeping for 1 second  
                            System.out.println("Thread " + Thread.currentThread().getName() + " at work.");  
                            Thread.sleep(1000);  
                        } catch (InterruptedException ie) {  
                            // ignore for now  
                        }  
                    }  
                });  
            }  
        } catch (RejectedExecutionException ree) {  
            // Let's see which task gets rejected  
            System.out.println("Task " + (i + 1) + " rejected.");  
        } finally {  
            // don't forget to shutdown the executor  
            threadPoolExecutor.shutdown();  
  
            // wait for the executor to shutdown  
            threadPoolExecutor.awaitTermination(1, TimeUnit.HOURS);  
        }  
    }  
}
```

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. In case the executor is shutdown then the task is simply discarded.

To demonstrate the `ThreadPoolExecutor.DiscardOldestPolicy` we'll create a class `MyTask` that extends `Runnable` to capture the order in which tasks are submitted to the thread. The program is similar to the previous examples and changes the rejection policy to `ThreadPoolExecutor.DiscardOldestPolicy`.

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

class Demonstration {

    // we create a class to capture the task number.
    static class MyTask implements Runnable {

        private int taskNum;

        public MyTask(int taskNum) {
            this.taskNum = taskNum;
        }

        @Override
        public void run() {
            try {
                // simulate work by sleeping for 1 seconds
                Thread.sleep(1000);
            } catch (InterruptedException ie) {
                // ignore
            }
            System.out.println("Hello this is thread " + Thread.currentThread().getName() + " executing task number " + taskNum);
        }
    }

    public static void main( String args[] ) throws InterruptedException {
        // create a ThreadPoolExecutor
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(1, 5, 1,
            TimeUnit.MINUTES, new LinkedBlockingDeque<>(5), new
        ThreadPoolExecutor.DiscardOldestPolicy());

        int i = 0;
```

```

try {
    // Try to submit 20 tasks
    for (; i < 20; i++) {
        threadPoolExecutor.execute(new MyTask(i + 1));
    }
} catch (RejectedExecutionException ree) {
    // Let's see which task gets rejected
    System.out.println("Task " + (i + 1) + " rejected.");
} finally {
    // don't forget to shutdown the executor
    threadPoolExecutor.shutdown();

    // wait for the executor to shutdown
    threadPoolExecutor.awaitTermination(1, TimeUnit.HOURS);
}
}
}
}

```

If you examine the output of the program above, you'll notice that the tasks that get executed aren't numbered sequentially. As threads become busy and the queue fills-up, the policy dictates to discard the oldest submitted task in the queue. Note that the last 5 tasks numbered 20, 19, 18, 17 and 16 are always executed as they are submitted in the end and there are no task submissions after them that may cause them to be discarded.

Shutting down#

The `ThreadPoolExecutor` can be shutdown by invoking the `shutdown()` method. A pool that is no longer referenced and has no remaining threads will shutdown automatically. In case `shutdown()` isn't invoked then the configuration must make sure that unused threads eventually die by setting the `corePoolSize` to zero and choosing an appropriate `keepAliveTime` value. Another option if `corePoolSize` is set to a non-zero is to use the `allowCoreThreadTimeOut(boolean)` method to have the time out policy apply to both core and non-core threads.

Hooks#

The `ThreadPoolExecutor` class also exposes protected overridable methods that derived classes can override. For instance: The `beforeExecute(Thread, Runnable)` and `afterExecute(Runnable, Throwable)` methods are called before and after execution of each task. These can be used to manipulate the execution environment; for example, reinitializing ThreadLocals, gathering statistics, or adding log entries etc. Similarly, the method `terminated()` can be overridden to perform any special processing

that needs to be done once the Executor has fully terminated. Note that if any of the `BlockingQueue` methods, callbacks or hooks throw an exception, threads in the pool may fail, terminate abruptly and possibly get replaced.

Conclusion#

In summary, for the vast majority of use-cases the `Executors` factory method should be used for instantiating thread pools, however, there may be instances where fine-grained control is desired and the `ThreadPoolExecutor` class is a good fit for such scenarios.

Callable Interface

This lesson discusses the Callable interface

Callable Interface#

In the previous sections we used the `Runnable` interface as the abstraction for tasks that were submitted to the executor service. The `Runnable` interface's sole `run` method doesn't return a value, which is a handicap for tasks that don't want to write results to global or shared datastructures. The interface `Callable` allows such tasks to return results. Let's see the definition of the interface first.

```
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

Note the interface also allows a task to throw an exception. A task goes through the various stages of its life which include the following:

- created
- submitted
- started
- completed

Let's say we want to compute the sum of numbers from 1 to n. Our task should accept an integer n and spit out the sum. Below are two ways to implement our task.

```
class SumTask implements Callable<Integer> {

    int n;

    public SumTask(int n) {
        this.n = n;
    }

    public Integer call() throws Exception {

        if (n <= 0)
            return 0;

        int sum = 0;
        for (int i = 1; i <= n; i++) {
            sum += i;
        }

        return sum;
    }
}
```

Or we could take advantage of the anonymous class feature in the Java language to declare our task like so:

```
final int n = 10
Callable<Integer> sumTask = new Callable<Integer>() {

    public Integer call() throws Exception {
        int sum = 0;
        for (int i = 1; i <= n; i++)
            sum += i;
        return sum;
    }
};
```

Now we know how to represent our tasks using the `Callable` interface. In the next section we'll explore the `Future` interface which will help us manage a task's lifecycle as well as retrieve results from it.

Future Interface

This lesson discusses the Future interface.

Future Interface#

The `Future` interface is used to represent the result of an asynchronous computation. The interface also provides methods to check the status of a submitted task and also allows the task to be cancelled if possible. Without further ado, let's dive into an example and see how callable and future objects work in tandem. We'll continue with our sumTask example from the previous lesson.

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {

    // Create and initialize a threadpool
    static ExecutorService threadPool = Executors.newFixedThreadPool(2);

    public static void main( String args[] ) throws Exception {
        System.out.println( "sum :" + findSum(10));
        threadPool.shutdown();
    }

    static int findSum(final int n) throws ExecutionException, InterruptedException {

        Callable<Integer> sumTask = new Callable<Integer>() {

            public Integer call() throws Exception {
                int sum = 0;
                for (int i = 1; i <= n; i++)
                    sum += i;
                return sum;
            }
        };

        Future<Integer> f = threadPool.submit(sumTask);
        return f.get();
    }

}
```

Using Future and Callable Together

Thread pools implementing the `ExecutorService` return a future for their task submission methods. In the above code on **line 29** we get back a future when submitting our task. We retrieve the result of the task by invoking the `get` method on the future. The `get` method will return the

result or throw an instance of `ExecutionException`. Let's see an example of that now.

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {

    static ExecutorService threadPool = Executors.newFixedThreadPool(2);

    public static void main( String args[] ) throws Exception {
        System.out.println( " sum: " + findSumWithException(10));
        threadPool.shutdown();
    }

    static int findSumWithException(final int n) throws ExecutionException, InterruptedException
    {

        int result = -1;

        Callable<Integer> sumTask = new Callable<Integer>() {

            public Integer call() throws Exception {
                throw new RuntimeException("something bad happened.");
            }
        };

        Future<Integer> f = threadPool.submit(sumTask);

        try {
            result = f.get();
        } catch (ExecutionException ee) {
            System.out.println("Something went wrong. " + ee.getCause());
        }

        return result;
    }
}
```

On **line 31** of the above code, we make a `get` method call. The method throws an execution exception, which we catch. The reason for the exception can be determined by using the `getCause` method of the execution

exception. If you run the above snippet, you'll see it prints the runtime exception that we throw on **line 24**.

The `get` method is a blocking call. It'll block till the task completes. We can also write a polling version, where we poll periodically to check if the task is complete or not. Future also allows us to cancel tasks. If a task has been submitted but not yet executed, then it'll be cancelled. However, if a task is currently running, then it may or may not be cancellable. We'll discuss cancelling tasks in detail in future lessons.

Below is an example where we create two tasks. We poll to check if the task has completed. Also, we cancel the second submitted task.

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {

    static ExecutorService threadPool = Executors.newSingleThreadExecutor();

    public static void main( String args[] ) throws Exception {
        System.out.println(pollingStatusAndCancelTask(10));
        threadPool.shutdown();
    }

    static int pollingStatusAndCancelTask(final int n) throws Exception {

        int result = -1;

        Callable<Integer> sumTask1 = new Callable<Integer>() {

            public Integer call() throws Exception {

                // wait for 10 milliseconds
                Thread.sleep(10);

                int sum = 0;
                for (int i = 1; i <= n; i++)
                    sum += i;
                return sum;
            }
        };

        Callable<Void> randomTask = new Callable<Void>() {
```

```

public Void call() throws Exception {

    // go to sleep for an hours
    Thread.sleep(3600 * 1000);
    return null;
}

};

Future<Integer> f1 = threadPool.submit(sumTask1);
Future<Void> f2 = threadPool.submit(randomTask);

// Poll for completion of first task
try {

    // Before we poll for completion of second task,
    // cancel the second one
    f2.cancel(true);

    // Polling the future to check the status of the
    // first submitted task
    while (!f1.isDone()) {
        System.out.println("Waiting for first task to complete.");
    }
    result = f1.get();
} catch (ExecutionException ee) {
    System.out.println("Something went wrong.");
}

System.out.println("\nls second task cancelled : " + f2.isCancelled());

return result;
}
}

```

```

43
44     .... Future<Integer> f1 = threadPool.submit(sumTask1);
45     .... Future<Void> f2 = threadPool.submit(randomTask);
46
47     .... // Poll for completion of first task
48     .... try {
49
50         .... // Before we poll for completion of second task,
51         .... // cancel the second one
52         .... f2.cancel(true);
53
54         .... // Polling the future to check the status of the
55         .... // first submitted task
56         .... while (!f1.isDone()) {
57             .... System.out.println("Waiting for first task to complete.");
58         }
59         .... result = f1.get();

```

Note the following about the above code

- On **lines 44 and 45** we submit two tasks for execution.
- **line 45** the second task submitted doesn't return any value so the future is parametrized with `Void`.
- On **line 52**, we cancel the second task. Since our thread pool consists of a single thread and the first task sleeps for a bit before it starts executing, we can assume that the second task will not have started executing and can be cancelled. This is verified by checking for and printing the value of the `isCancelled` method later in the program.
- On **lines 56 - 58**, we repeatedly poll for the status of the first task.

The final output of the program shows messages from polling and the status of the second task cancellation request.

FutureTask#

Java also provides an implementation of the future interface called the `FutureTask`. It can wrap a callable or runnable object and in turn be submitted to an executor. Though, the class may not be very useful if you don't intend to create customized tasks but we mention it for the sake of completeness.

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.FutureTask;

class Demonstration {

    @SuppressWarnings("unchecked")
    public static void main( String args[] ) throws Exception{

        FutureTask<Integer> futureTask = new FutureTask<new Callable() {

            public Object call() throws Exception {
                try{
                    Thread.sleep(1);
                }
                catch(InterruptedException ie){
                    // swallow exception
                }
            }
        }
    }
}
```

```

        return 5;
    }
});

ExecutorService threadPool = Executors.newSingleThreadExecutor();
Future duplicateFuture = threadPool.submit(futureTask);

// Awful idea to busy wait
while (!futureTask.isDone()) {
    System.out.println("Waiting");
}

if(duplicateFuture.isDone() != futureTask.isDone()){
    System.out.println("This should never happen.");
}

System.out.println((int)futureTask.get());

threadPool.shutdown();
}
}

```

CompletionService Interface

This lesson talks about how to batch multiple tasks together

CompletionService Interface#

In the previous lesson we discussed how tasks can be submitted to executors but imagine a scenario where you want to submit hundreds or thousands of tasks. You'll retrieve the future objects returned from the submit calls and then poll all of them in a loop to check which one is done and then take appropriate action. Java offers a better way to address this use case through the **CompletionService** interface. You can use the **ExecutorCompletionService** as a concrete implementation of the interface.

The completion service is a combination of a blocking queue and an executor. Tasks are submitted to the queue and then the queue can be polled for completed tasks. The service exposes two methods, one **poll** which returns null if no task is completed or none were submitted and two **take** which blocks till a completed task is available.

Below is an example program that demonstrates the use of completion service.

```
import java.util.Random;
```

```
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {

    static Random random = new Random(System.currentTimeMillis());

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

    static void completionServiceExample() throws Exception {

        class TrivialTask implements Runnable {

            int n;

            public TrivialTask(int n) {
                this.n = n;
            }

            public void run() {
                try {
                    // sleep for one second
                    Thread.sleep(random.nextInt(101));
                    System.out.println(n*n);
                } catch (InterruptedException ie) {
                    // swallow exception
                }
            }
        }

        ExecutorService threadPool = Executors.newFixedThreadPool(3);
        ExecutorCompletionService<Integer> service =
            new ExecutorCompletionService<Integer>(threadPool);

        // Submit 10 trivial tasks.
        for (int i = 0; i < 10; i++) {
            service.submit(new TrivialTask(i), new Integer(i));
        }

        // wait for all tasks to get done
        int count = 10;
        while (count != 0) {
```

```

        Future<Integer> f = service.poll();
        if (f != null) {
            System.out.println("Thread" + f.get() + " got done.");
            count--;
        }
    }

    threadPool.shutdown();
}

}

```

ThreadLocal

This lesson discusses thread local storage

ThreadLocal#

Consider the following instance method of a class

```

void add(int val) {

    int count = 5;
    count += val;
    System.out.println(val);

}

```

Do you think the above method is thread-safe? If multiple threads call this method, then each executing thread will create a copy of the local variables on its own thread stack. There would be no shared variables amongst the threads and the instance method by itself would be thread-safe.

However, if we moved the `count` variable out of the method and declared it as an instance variable then the same code will not be thread-safe.

We can have a copy of an instance (or a class) variable for each thread that accesses it by declaring the instance variable *ThreadLocal*. Look at the thread unsafe code below. If you run it multiple times, you'll see different results. The count variable is incremented 100 times by 100 threads so in a thread-safe world the final value of the variable should come out to be 10,000.

```

import java.util.concurrent.Executors;

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

```

```

UnsafeCounter usc = new UnsafeCounter();
Thread[] tasks = new Thread[100];

for (int i = 0; i < 100; i++) {
    Thread t = new Thread(() -> {
        for (int j = 0; j < 100; j++)
            usc.increment();
    });
    tasks[i] = t;
    t.start();
}

for (int i = 0; i < 100; i++) {
    tasks[i].join();
}

System.out.println(usc.count);
}
}

```

```

class UnsafeCounter {

    // Instance variable
    int count = 0;

    void increment() {
        count = count + 1;
    }
}

```

Now we'll change the code to make the instance variable threadlocal. The change is:

```
ThreadLocal<Integer> counter = ThreadLocal.withInitial(() -> 0);
```

The above code creates a separate and completely independent copy of the variable `counter` for every thread that calls the `increment()` method. Conceptually, you can think of a **ThreadLocal<T>** variable as a map that contains mapping for each thread and its copy of the threadlocal variable or equivalently a **Map<Thread, T>**. Though this is not how it is actually implemented. Furthermore, the thread specific values are stored in the thread object itself and are eligible for garbage collection once a thread terminates (if no other references exist to the threadlocal value).

The code below is a fixed version of the unsafe counter. Note that each thread has its own copy of the **counter** variable, which is incremented a 100 times. Therefore, each thread increments its own copy of counter a 100 times and that value gets printed for each thread.

ThreadLocal variables get tricky when used with the executor service (threadpools) since threads that don't terminate aren't returned to the threadpool. So any threadlocal variables for such threads aren't garbage collected. For interesting scenarios, please see Quiz#8.

```
class Demonstration {  
    public static void main( String args[] ) throws Exception{  
        UnsafeCounter usc = new UnsafeCounter();  
        Thread[] tasks = new Thread[100];  
  
        for (int i = 0; i < 100; i++) {  
            Thread t = new Thread(() -> {  
                for (int j = 0; j < 100; j++)  
                    usc.increment();  
  
                System.out.println(usc.counter.get());  
            });  
            tasks[i] = t;  
            t.start();  
        }  
  
        for (int i = 0; i < 100; i++) {  
            tasks[i].join();  
        }  
        System.out.println(usc.counter.get());  
    }  
}  
  
class UnsafeCounter {  
  
    ThreadLocal<Integer> counter = ThreadLocal.withInitial(() -> 0);  
  
    void increment() {  
        counter.set(counter.get() + 1);  
    }  
}
```

ThreadLocalRandom

Guide to using ThreadLocalRandom with examples.

Overview#

The class `java.util.concurrent.ThreadLocalRandom` is derived from `java.util.Random` and generates random numbers much more efficiently than `java.util.Random` in multithreaded scenarios. Interestingly, `Random` is *thread-safe* and can be used by multiple threads without malfunction, just not efficiently.

To understand why an instance of the `Random` class experiences overhead and contention in concurrent programs, we'll delve into the code for one of the most commonly used methods `nextInt()` of the `Random` class. The code is reproduced verbatim from the Java source code below

```
/**  
 * Generates the next pseudorandom number. Subclasses should  
 * override this, as this is used by all other methods.  
 *  
 * <p>The general contract of {@code next} is that it returns an  
 * {@code int} value and if the argument {@code bits} is between  
 * {@code 1} and {@code 32} (inclusive), then that many low-order  
 * bits of the returned value will be (approximately) independently  
 * chosen bit values, each of which is (approximately) equally  
 * likely to be {@code 0} or {@code 1}. The method {@code next} is  
 * implemented by class {@code Random} by atomically updating the seed  
 * to  
 * <pre>{@code (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1)}</pre>  
 * and returning  
 * <pre>{@code (int)(seed >>> (48 - bits))}</pre>  
 *  
 * This is a linear congruential pseudorandom number generator, as  
 * defined by D. H. Lehmer and described by Donald E. Knuth in  
 * <i>The Art of Computer Programming, Volume 2:  
 * <i>Seminumerical Algorithms</i>, section 3.2.1.  
 *  
 * @param bits random bits  
 * @return the next pseudorandom value from this random number  
 *         generator's sequence  
 * @since 1.1  
 */  
protected int next(int bits) {  
    long oldseed, nextseed;  
    AtomicLong seed = this.seed;  
    do {  
        oldseed = seed.get();  
        nextseed = (oldseed * multiplier + addend) & mask;  
    } while (!seed.compareAndSet(oldseed, nextseed));  
    return (int)(nextseed >>> (48 - bits));  
}
```

Examine the code above and realize the `do-while` loop uses the `compareAndSet()` method to atomically set the `seed` variable to a new value in its predicate. Imagine several threads invoking the `next()` method on a shared instance of `Random`, only one thread will successfully exit the loop and

the rest will re-execute the loop, until all of them exit one by one. This mechanism to update the `seed` variable is precisely what makes the `Random` class inefficient for highly concurrent programs, when several threads want to generate random numbers in parallel.

The performance issues faced by `Random` are addressed by the `ThreadLocalRandom` class which is isolated in its effects to a single thread. A random number generated by one thread using `ThreadLocalRandom` has no bearing on random numbers generated by other threads, unlike an instance of `Random` that generates random numbers globally. Furthermore, `ThreadLocalRandom` differs from `Random` in that the former doesn't allow setting a seed value unlike the latter. In summary, `ThreadLocalRandom` is more performant than `Random` as it eliminates concurrent access to shared state.

The astute reader would question if maintaining a distinct `Random` object per thread is equivalent to using the `ThreadLocalRandom` class? The `ThreadLocalRandom` class is singleton and uses state held by the `Thread` class to generate random numbers. In particular the `Thread` class houses the following fields for `ThreadLocalRandom` to use for generating random numbers and related book-keeping.

```
class Thread implements Runnable {

    // The following three initially uninitialized fields are exclusively
    // managed by class java.util.concurrent.ThreadLocalRandom. These
    // fields are used to build the high-performance PRNGs in the
    // concurrent code, and we can not risk accidental false sharing.
    // Hence, the fields are isolated with @Contended.

    /** The current seed for a ThreadLocalRandom */
    @jdk.internal.vm.annotation.Contended("tlr")
    long threadLocalRandomSeed;

    /** Probe hash value; nonzero if threadLocalRandomSeed initialized */
    @jdk.internal.vm.annotation.Contended("tlr")
    int threadLocalRandomProbe;

    /** Secondary seed isolated from public ThreadLocalRandom sequence */
    @jdk.internal.vm.annotation.Contended("tlr")
    int threadLocalRandomSecondarySeed;

}
```

Each thread stores the seed itself in the field `threadLocalRandomSeed`. As the seed is not shared among threads anymore, performance improves.

Usage#

The idiomatic usage for generating random numbers takes the form of `ThreadLocalRandom.current().nextInt()` and is demonstrated in the widget below:

```
import java.util.concurrent.ThreadLocalRandom;

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

        // generate a random boolean value
        System.out.println(ThreadLocalRandom.current().nextBoolean());

        // generate a random int value
        System.out.println(ThreadLocalRandom.current().nextInt());

        // generate a random int between 0 (inclusive) and 500 (exclusive)
        System.out.println(ThreadLocalRandom.current().nextInt(500));

        // generate a random int between 700 (inclusive) and 1900 (exclusive)
        System.out.println(ThreadLocalRandom.current().nextInt(700,1900));

        // generate a random double between 0 (inclusive) and 1 (exclusive)
        System.out.println(ThreadLocalRandom.current().nextDouble());

        // generate a random float value between 0 (inclusive) and 1 (exclusive)
        System.out.println(ThreadLocalRandom.current().nextFloat());

        // generate a Gaussian ("normally") distributed double value with mean 0.0 and
        // standard deviation 1.0 from this random number generator's sequence
        System.out.println(ThreadLocalRandom.current().nextGaussian());
    }
}
```

Difference

between **Random** and **ThreadLocalRand om#**

Consider the scenario of a single instance of `Random` class being shared among 5 threads. Our program has each thread generate a random integer ten thousand times. We repeat the same test using the `ThreadLocalRandom` class and time the execution for both scenarios in milliseconds. As expected, the test using the `ThreadLocalRandom` class performs better than the one using the `Random` class instance. Though our test is crude but it still gives us a sense of difference in performance of the two classes.

Some runs of the program may exhibit a longer execution time for `ThreadLocalRandom` class than the `Random` class. This may occur due to the widget code executing in a shared cloud environment beyond our control. However, if the reader executed the code with all aspects as constants `ThreadLocalRandom` would outperform `Random` when generating random numbers in our text code.

```
import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ThreadLocalRandom;

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

        performanceUsingRandom();
        performanceUsingThreadLocalRandom();
    }

    static void performanceUsingThreadLocalRandom() throws Exception {

        ExecutorService es = Executors.newFixedThreadPool(15);

        Runnable task = new Runnable() {
            @Override
            public void run() {

                for (int i = 0; i < 50000; i++) {
                    ThreadLocalRandom.current().nextInt();
                }
            }
        };
    };

    int numThreads = 4;
    Future[] futures = new Future[numThreads];
    long start = System.currentTimeMillis();

    try {
        for (int i = 0; i < numThreads; i++)
            futures[i] = es.submit(task);

        for (int i = 0; i < numThreads; i++)
            futures[i].get();
    }
```

```

        long executionTime = System.currentTimeMillis() - start;
        System.out.println("Execution time using ThreadLocalRandom : " + executionTime + "
milliseconds");

    } finally {
        es.shutdown();
    }
}

static void performanceUsingRandom() throws Exception {

    Random random = new Random();
    ExecutorService es = Executors.newFixedThreadPool(15);

    Runnable task = new Runnable() {
        @Override
        public void run() {

            for (int i = 0; i < 50000; i++){
                random.nextInt();
            }
        }
    };
}

int numThreads = 4;
Future[] futures = new Future[numThreads];
long start = System.currentTimeMillis();

try {
    for (int i = 0; i < numThreads; i++)
        futures[i] = es.submit(task);

    for (int i = 0; i < numThreads; i++)
        futures[i].get();

    long executionTime = System.currentTimeMillis() - start;
    System.out.println("Execution time using Random : " + executionTime + " milliseconds");

} finally {
    es.shutdown();
}
}

```

CountDownLatch

Explanation#

`CountDownLatch` is a synchronization primitive that comes with the `java.util.concurrent` package. It can be used to block a single or multiple threads while other threads complete their operations.

A `CountDownLatch` object is initialized with the number of tasks/threads it is required to wait for. Multiple threads can block and wait for the `CountDownLatch` object to reach zero by invoking `await()`. Every time a thread finishes its work, the thread invokes `countDown()` which decrements the counter by 1. Once the count reaches zero, threads waiting on the `await()` method are notified and resume execution.

The counter in the `CountDownLatch` cannot be reset making the `CountDownLatch` object unreusable. A `CountDownLatch` initialized with a count of 1 serves as an on/off switch where a particular thread is simply waiting for its only partner to complete. Whereas a `CountDownLatch` object initialized with a count of N indicates a thread waiting for N threads to complete their work. However, a single thread can also invoke `countDown()` N times to unblock a thread more than once.

If the `CountDownLatch` is initialized with zero, the thread would not wait for any other thread(s) to complete. The count passed is basically the number of times `countDown()` must be invoked before threads can pass through `await()`. If the `CountDownLatch` has reached zero and `countDown()` is again invoked, the latch will remain released hence making no difference.

A thread blocked on `await()` can also be interrupted by another thread as long as it is waiting and the counter has not reached zero.

Let's take an example where a master thread waits for worker threads to complete their execution.

Two workers, A & B, are being executed concurrently (two back to back threads initiated) while the master thread waits for them to finish. Every time a worker completes execution, the counter in the `CountDownLatch` is decremented by 1. Once all the workers have completed execution, the counter reaches 0 and notifies the threads blocked on the `await()` method. Subsequently, the latch opens and allows the master thread to run.

```
/**  
 * The worker thread that has to complete its tasks first  
 */
```

```

public class Worker extends Thread
{
    private CountDownLatch countDownLatch;

    public Worker(CountDownLatch countDownLatch, String name) {
        super(name);
        this.countDownLatch = countDownLatch;
    }

    @Override
    public void run()
    {
        System.out.println("Worker " +Thread.currentThread().getName()+" started");
        try
        {
            Thread.sleep(3000);
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
        System.out.println("Worker "+Thread.currentThread().getName()+" finished");

        //Each thread calls countDown() method on task completion.
        countDownLatch.countDown();
    }
}

/**
 * The master thread that has to wait for the worker to complete its operations first
 */
public class Master extends Thread
{
    public Master(String name)
    {
        super(name);
    }

    @Override
    public void run()
    {
        System.out.println("Master executed "+Thread.currentThread().getName());
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
    }
}

```

```


/**
 * The main thread that executes both the threads in a particular order
 */
public class Main
{
    public static void main(String[] args) throws InterruptedException
    {
        //Created CountDownLatch for 2 threads
        CountDownLatch countDownLatch = new CountDownLatch(2);

        //Created and started two threads
        Worker A = new Worker(countDownLatch, "A");
        Worker B = new Worker(countDownLatch, "B");

        A.start();
        B.start();

        //When two threads(A and B)complete their tasks, they are returned (counter reached 0).
        countDownLatch.await();

        //Now execution of master thread has started
        Master D = new Master("Master executed");
        D.start();
    }
}


```

Main.java

```

import java.util.concurrent.CountDownLatch;
public class main
{
    public static void main(String[] args) throws InterruptedException
    {

        //Created CountDownLatch for 2 threads
        CountDownLatch countDownLatch = new CountDownLatch(2);

        //Created and started two threads
        Worker A = new Worker(countDownLatch, "A");
        Worker B = new Worker(countDownLatch, "B");

        A.start();
        B.start();
        countDownLatch.countDown();

        //When two threads(A and B)complete their tasks, they are returned (counter reached 0).
    }
}

```

```

        countDownLatch.await();

        //Now execution of master thread has started
        Master D = new Master("");
        D.start();
    }
}

```

Worker.java

```

import java.util.concurrent.CountDownLatch;
/**
 * The manufacturer thread that has to complete its tasks first
 */
public class Worker extends Thread
{
    private CountDownLatch countDownLatch;

    public Worker(CountDownLatch countDownLatch, String name) {
        super(name);
        this.countDownLatch = countDownLatch;
    }

    @Override
    public void run()
    {
        System.out.println("Worker " +Thread.currentThread().getName()+" started");
        try
        {
            Thread.sleep(3000);
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
        System.out.println("Worker "+Thread.currentThread().getName()+" finished");

        //Each thread calls countDown() method on task completion.
        countDownLatch.countDown();
    }
}

```

Master.java

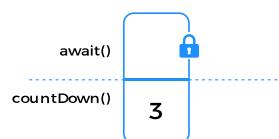
```
public class Master extends Thread
{
    public Master(String name)
    {
        super(name);
    }

    @Override
    public void run()
    {
        System.out.println("Master executed "+Thread.currentThread().getName());
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

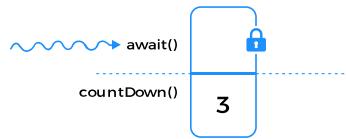
A pictorial representation appears below:

CountDownLatch

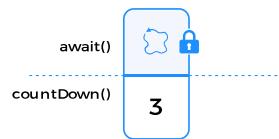
1. CountDownLatch initialized with a count of 3



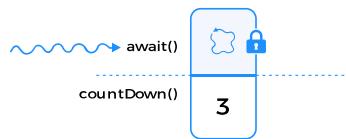
2. A thread invokes await() on the CountDownLatch object



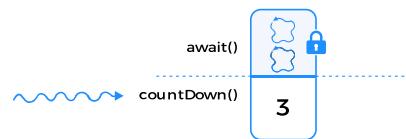
3. Thread is blocked till the count reaches 0



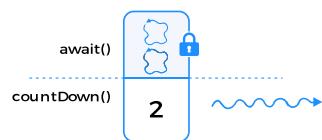
4. A second thread invokes await() and gets blocked



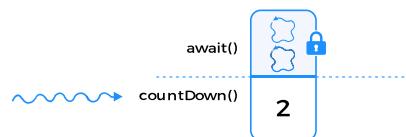
5. Two threads waiting for count to reach 0 and another thread invokes countDown()



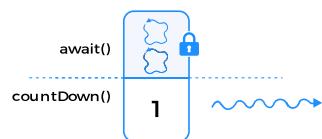
6. Count is now 2



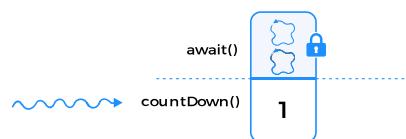
7. Another thread invokes countDown()



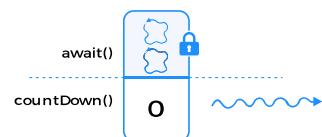
8. Count is decremented to 1



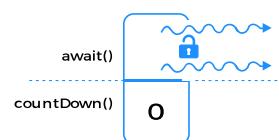
9. A third thread invokes countDown()



10. Count reaches 0



11. Latch opens and the two blocked threads are allowed to proceed.



CyclicBarrier

Explanation#

`CyclicBarrier` is a synchronization mechanism introduced in JDK 5 in the `java.util.concurrent` package. It allows multiple threads to wait for each other at a common point (barrier) before continuing execution. The threads wait for each other by calling the `await()` method on the `CyclicBarrier`. All threads that wait for each other to reach barrier are called parties. You can *read-up on designing and implementing a barrier for an interview question [here](#).*

`CyclicBarrier` is initialized with an integer that denotes the number of threads that need to call the `await()` method on the barrier. Second argument in `CyclicBarrier`'s constructor is a `Runnable` instance that includes the action to be executed once the last thread arrives.

The most useful property of `CyclicBarrier` is that it can be reset to its initial state by calling the `reset()` method. It can be reused after all the threads have been released.

Lets take an example where `CyclicBarrier` is initialized with 3 worker threads that will have to cross the barrier. All the threads need to call the `await()` method. Once all the threads have reached the barrier, it gets broken and each thread starts its execution from that point onwards.

```
/*
 * Runnable task for each thread.
 */
class Task implements Runnable {

    private CyclicBarrier barrier;

    public Task(CyclicBarrier barrier) {
        this.barrier = barrier;
    }

    //Await is invoked to wait for other threads
    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " is waiting on barrier");
            barrier.await();
            //printing after crossing the barrier
            System.out.println(Thread.currentThread().getName() + " has crossed the barrier");
        } catch (InterruptedException ex) {
            Logger.getLogger(Task.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

```

        } catch (BrokenBarrierException ex) {
            Logger.getLogger(Task.class.getName()).log(Level.SEVERE, null
, ex);
        }
    }

}

/**
 * Main thread that demonstrates how to use CyclicBarrier.
 */
public class Main {
    public static void main (String args[]) {

        //Creating CyclicBarrier with 3 parties i.e. 3 Threads needs to call await()
        final CyclicBarrier cb = new CyclicBarrier(3, new Runnable(){

            //Action that executes after the last thread arrives
            @Override
            public void run(){
                System.out.println("All parties have arrived at the barrier, lets continue execution.");
            }
        });

        //starting each thread
        Thread t1 = new Thread(new Task(cb), "Thread 1");
        Thread t2 = new Thread(new Task(cb), "Thread 2");
        Thread t3 = new Thread(new Task(cb), "Thread 3");

        t1.start();
        t2.start();
        t3.start();
    }
}

```

Main.java

```

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

/**
 * Main thread that demonstrates how to use CyclicBarrier.
 */
public class main {
    public static void main (String args[]) {

```

```

//Creating CyclicBarrier with 3 parties i.e. 3 Threads needs to call await()
final CyclicBarrier cb = new CyclicBarrier(3, new Runnable(){

    //Action that executes after the last thread arrives
    @Override
    public void run(){
        //This task will be executed once all threads reaches barrier
        System.out.println("All parties have arrived at the barrier, lets continue execution.");
    }
});

//starting each thread
Thread t1 = new Thread(new Task(cb), "Thread 1");
Thread t2 = new Thread(new Task(cb), "Thread 2");
Thread t3 = new Thread(new Task(cb), "Thread 3");

t1.start();
t2.start();
t3.start();
}
}

```

Task.java

```

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * Runnable task for each thread.
 */
class Task implements Runnable {

    private CyclicBarrier barrier;

    public Task(CyclicBarrier barrier) {
        this.barrier = barrier;
    }

    //Await is invoked to wait for other threads
    @Override
    public void run() {

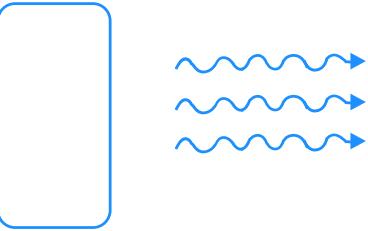
```

```
try {  
    System.out.println(Thread.currentThread().getName() + " is waiting on barrier");  
    barrier.await();  
    //printing after crossing the barrier  
    System.out.println(Thread.currentThread().getName() + " has crossed the barrier");  
} catch (InterruptedException ex) {  
    Logger.getLogger(Task.class.getName()).log(Level.SEVERE, null, ex);  
} catch (BrokenBarrierException ex) {  
    Logger.getLogger(Task.class.getName()).log(Level.SEVERE, null, ex);  
}  
}  
}
```

A pictorial representation appears below:

Working of a Barrier

1. No thread has reached the barrier yet

Size = 3
2. The first thread reaching the barrier is blocked
3. A second thread making its way to the barrier
4. Two threads waiting at the barrier for a third one to arrive
5. All threads reach the barrier
6. The barrier releases all threads

Concurrent Collections

This lesson gives a brief introduction about Java's concurrent collection classes.

Concurrent Collections#

A bit of history before we delve into concurrent collection classes offered by Java. When the Collections Framework was introduced in JDK 1.2, it didn't come with collections that were synchronized. However, to cater for multithreaded scenarios, the framework provided static methods to wrap vanilla collections in thread-safe wrapper objects. These thread-safe wrapper objects came to be known as *wrapper collections*.

Example Wrapper Collection

```
ArrayList<Integer> myList = new ArrayList<>();  
List<Integer> syncList = Collections.synchronizedList(myList);
```

For design pattern fans, this is an example of the *decorator pattern*.

Java 5 introduced thread-safe concurrent collections as part of a much larger set of concurrency utilities. The concurrent collections remove the necessity for client-side locking. In fact, external synchronization is not even possible with these collections, as there is no one object which when locked will synchronize all instance methods. If you need thread safety, the concurrent collections generally provide much better performance than synchronized (wrapper) collections. This is primarily because their throughput is not reduced by the need to serialize access, as is the case with synchronized collections. Synchronized collections also suffer from the overhead of managing locks, which can be high if there is much contention.

The concurrent collections use a variety of ways to achieve thread-safety while avoiding traditional synchronization for better performance. These are:

- **Copy on Write:** Concurrent collections utilizing this scheme are suitable for read-heavy use cases. An immutable copy is created of the backing collection and whenever a write operation is attempted, the copy is discarded and a new copy with the change is created. Reads of the collection don't require any synchronization, though synchronization is needed briefly when the new array is being created. Examples include `CopyOnWriteArrayList` and `CopyOnWriteArraySet`

- **Compare and Swap:** Consider a computation in which the value of a single variable is used as input to a long-running calculation whose eventual result is used to update the variable. Traditional synchronization makes the whole computation atomic, excluding any other thread from concurrently accessing the variable. This reduces opportunities for parallel execution and hurts throughput. An algorithm based on CAS behaves differently: it makes a local copy of the variable and performs the calculation without getting exclusive access. Only when it is ready to update the variable does it call CAS, which in one atomic operation compares the variable's value with its value at the start and, if they are the same, updates it with the new value. If they are not the same, the variable must have been modified by another thread; in this situation, the CAS thread can try the whole computation again using the new value, or give up, or—in some algorithms—continue, because the interference will have actually done its work for it! Collections using CAS include `ConcurrentLinkedQueue` and `ConcurrentSkipListMap`.
- **Lock:** Some collection classes use `Lock` to divide up the collection into multiple parts that can be locked separately resulting in improved concurrency. For example, `LinkedBlockingQueue` has separate locks for the head and tail ends of the queue, so that elements can be added and removed in parallel. Other collections using these locks include `ConcurrentHashMap` and most of the implementations of `BlockingQueue`.

CopyOnWrite Example#

Lets take an example with a regular `ArrayList` along with a `CopyOnWriteArrayList`. We will measure the time it takes to add an item to an already initialized array. The output from running the code widget below demonstrates that the `CopyOnWriteArrayList` takes much more time than a regular `ArrayList` because under the hood, all the elements of the `CopyOnWriteArrayList` object get copied thus making an insert operation that much more expensive.

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

/**
 * Java program to illustrate CopyOnWriteArrayList
 */
public class main
{
    public static void main(String[] args)
        throws InterruptedException
```

```

{
    //Initializing a regular ArrayList
    ArrayList<Integer> array_list = new ArrayList<>();
    array_list.ensureCapacity(500000);
    //Initializing a new CopyOnWrite ArrayList with 500,000 numbers
    CopyOnWriteArrayList<Integer> numbers = new CopyOnWriteArrayList<>(array_list);

    //Calculating the time it takes to add a number in CopyOnWrite ArrayList
    long startTime = System.nanoTime();
    numbers.add(500001);
    long endTime = System.nanoTime();
    long duration = (endTime - startTime);

    //Calculating the time it takes to add a number in regular ArrayList
    long startTime_al = System.nanoTime();
    array_list.add(500001);
    long endTime_al = System.nanoTime();
    long duration_al = (endTime_al - startTime_al);

    System.out.println("Time taken by a regular arraylist: " + duration_al + " nano seconds");
    System.out.println("Time taken by a CopyOnWrite arraylist: " + duration + " nano seconds");
}

}

```

ConcurrentHashMap

This lesson explains the working of the concurrent hash map and the other hash map data structures available in Java.

We'll cover the following



- HashMaps and Concurrency
- Using ConcurrentHashMap
 - Properties of ConcurrentHashMap
 - Newbie Mistakes with ConcurrentHashMap
 - Fixing with Atomic Integer
 - Fixing with Custom Counter Class
- HashMap vs HashTable vs ConcurrentHashMap
- Performance

HashMaps and Concurrency#

HashMap is a commonly used data structure offering constant time access, however, it is not thread-safe. Consider the two methods `get()` and `put()` that get invoked by two different threads on an instance of `HashMap` in the following sequence:

1. Thread 1 invokes `put()` and inserts the key value pair ("myKey", "item-1")
2. Thread 2 invokes `get()` but before get operation completes, the thread is context-switched
3. Thread 1 updates myKey with a new value say "item-2"
4. Thread 2 becomes active again but retrieves the stale key value pair ("myKey", "item-1")

The above scenario is naive and one may argue that if stale values are acceptable for an application then we may get away without using synchronization. However, that is not true. Consider the scenario when a `HashMap` reaches capacity and resizes. When a resize occurs the elements from the old structure are copied over to the new structure and a rehashing

of the elements that hash to the same bucket and form a collision list may take place to reduce the size of the collision list. As one can imagine, there are several things that can go wrong when multiples threads operate on a map whilst the resize takes place. For instance, a reader thread might be iterating over the map when a rehashing occurs and a key that is present in the map is erroneously reported to be not present, since it was rehashed to a different bucket. When we throw into the mix all the other operation such as `remove` and `putIfAbsent`, it is easy to come up with sequences that demonstrate why `HashMap` is inherently thread unsafe. Compared to its synchronized cousins `HashTable` and `ConcurrentHashMap`, `HashMap` is lightweight and faster, but can only be used in single-threaded scenarios.

Using ConcurrentHashMap#

`ConcurrentHashMap` is a thread-safe class and multiple threads can operate on it in parallel without incurring any of the issues that a `HashMap` may suffer from in a concurrent environment. For write operations the entire map is never locked rather only a segment of the map is locked. However, the retrieval or read operations generally don't involve locking at all. So in case of a read, the value set for a key by the most recently completed update operation is returned i.e. a completed update operation on a given key bears a *happens before* relationship with any (non-null) read operation. This does mean that a stale value may be returned if an update operation is in progress but not yet completed.

Since read operations can happen while update operations are on-going, any concurrent reads during the execution of aggregate operations such as `putAll()` or `clear()` may return insertion or removal of some of the entries respectively, when all or none are expected.

Another important detail is that `Iterators`, `SplitIterator`, or `Enumerations` for an instance of the `ConcurrentHashMap` represent the state or snapshot of the data structure at a point in time, specifically when they were created and don't throw the `ConcurrentModificationException` exception. Though the iterator itself is designed to be used by a single thread. See next lesson for details on `ConcurrentModificationException`.

Properties of ConcurrentHashMap#

Other notable properties of the `ConcurrentHashMap` class are listed below:

- `null` can't be inserted either as a key or a value.
- The `ConcurrentHashMap` shards its data into segments and the segments are locked individually when being written to. Each segment can be

written independently of other segments allowing multiple threads to operate on the map object.

- The reads happen without locking for the majority of cases, thus making them synchronization-free and improving performance. However, note that there are certain minority scenarios when reads have to go through synchronization.
- In general, using keys that evaluate to the same hashCode will slow down the performance of any hash map.

Newbie Mistakes with ConcurrentHashMap#

One of the follies assumed when working with `ConcurrentHashMap` is to think that any accesses of and operations on the key/values within the data structure are somehow magically thread-safe. The map doesn't protect against external race conditions. Consider the program in the widget below that has two threads increment a key's value in a `ConcurrentHashMap` by a hundred times each. Run the program multiple times and see each run prints a different result.

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {

    public static void main( String args[] ) throws Exception {
        ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
        map.put("Biden", 0);

        ExecutorService es = Executors.newFixedThreadPool(5);

        // create a task to increment the vote count
        Runnable task = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 100; i++)
                    map.put("Biden", map.get("Biden") + 1);
            }
        };

        // submit the task twice
        Future future1 = es.submit(task);
        Future future2 = es.submit(task);

        // wait for the threads to finish
    }
}
```

```

future1.get();
future2.get();

// shutdown the executor service
es.shutdown();

System.out.println("votes for Biden = " + map.get("Biden"));
}
}

```

We sort of cheated in the above program to make it fail. Consider the line:

```
map.put("Biden", map.get("Biden") + 1)
```

The above line is really three operations:

1. retrieval of the value
2. incrementing the value
3. updating the value

The right implementation should execute all the three steps together as a transaction or atomically to avoid synchronization issues. The takeaway is that a `ConcurrentHashMap` doesn't protect its constituents from race conditions but access to the data structure itself is thread safe.

Fixing with Atomic Integer#

One of the ways to fix the above program is to use instance of the `AtomicInteger` class as value. We'll invoke the `incrementAndGet()` method to register an increment on the value. The code is presented below:

```

import java.util.concurrent.ConcurrentHashMap;
import java.util.Map;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.atomic.AtomicInteger;

class Demonstration {

    public static void main( String args[] ) throws Exception {
        ConcurrentHashMap<String, AtomicInteger> map = new ConcurrentHashMap<>();
        // create an atomic integer to keep the vote count
        AtomicInteger ai = new AtomicInteger(0);
        map.put("Biden", ai);
    }
}

```

```

ExecutorService es = Executors.newFixedThreadPool(5);

// create a task to increment the vote count
Runnable task = new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            // We are ignoring the returned updated value from the
            // function call
            map.get("Biden").incrementAndGet();
        }
    };
}

// submit the task twice
Future future1 = es.submit(task);
Future future2 = es.submit(task);

// wait for the threads to finish
future1.get();
future2.get();

// shutdown the executor service
es.shutdown();

System.out.println("votes for Biden = " + map.get("Biden").get());
}
}

```

The count of the above program will always be 200 no matter how many times you run it. But this brings up another question: if we use `AtomicInteger` as value with the `HashMap` class would our program output the correct result? The answer is yes for this naive/simple program because the atomic integer itself is thread-safe so multiple threads attempting to increment it do so serially. However, the data structure i.e. the hash map itself is thread-unsafe and can exhibit concurrency bugs when multiple threads operate on it, traverse its keys or values, or when the map resizes. Remember, we have to think about concurrency both at the map level and at the key/value level.

Fixing with Custom Counter Class#

We could re-write the above program with a class that tracks the count and perform explicit synchronization ourselves instead of relying on the `AtomicInteger` class. Read the listing below:

```

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {

    // Class to keep track of vote count
    static class MyCounter {
        private int count = 0;

        void increment() {
            count++;
        }

        int getCount() {
            return count;
        }
    }

    public static void main( String args[] ) throws Exception {
        ConcurrentHashMap<String, MyCounter> map = new ConcurrentHashMap<>();
        map.put("Biden", new MyCounter());

        ExecutorService es = Executors.newFixedThreadPool(5);

        // create a task to increment the vote count
        Runnable task = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    MyCounter mc = map.get("Biden");

                    // explicit synchronization
                    synchronized (mc) {
                        mc.increment();
                    }
                }
            }
        };
    }

    // submit the task twice
    Future future1 = es.submit(task);
    Future future2 = es.submit(task);

    // wait for the threads to finish
    future1.get();
    future2.get();
}
```

```

    // shutdown the executor service
    es.shutdown();

    System.out.println("votes for Biden = " + map.get("Biden").getCount());
}
}

```

HashMap vs HashTable vs ConcurrentHashMap#

As a Java newbie when looking for an appropriate choice for a hash map you'll be confronted with the following options:

1. `HashMap`
2. `Hashtable`
3. `Collections.synchronizedMap(...)`
4. `ConcurrentHashMap`

As already discussed `HashMap` isn't thread-safe and if your scenario doesn't involve multiple threads, pick `HashMap` as it will provide the best performance. `Hashtable` is a synchronized map i.e. it can only be accessed by a single reader or writer thread at a time. `Hashtable` performs poorly in a highly concurrent environment. Iterators on a `Hashtable` observe *fail fast* behavior i.e. the iterators throw a `ConcurrentModificationException` if the `Hashtable` is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` method. `Hashtable` is part of Java's legacy code and was retrofitted to implement the `Map` interface so that it could become part of the Java's collections framework.

`Collections.synchronizedMap(...)` creates an instance of the `SynchronizedMap` class (a private class in `Collections` class) backed by the passed-in map. The returned object synchronizes access to the backing map allowing a single thread to operate on the map at a time. For example:

```
Map<String, Integer> map = Collections.synchronizedMap(new HashMap<>());
```

The user must synchronize manually on the returned map object (not the backing map) when traversing any of the collection views using iterators, split iterators or streams.

A difference between `HashTable` and `Collections.synchronizedMap(...)` is that in case of `HashTable` all the methods are synchronized on the map object itself i.e. the `this` object. In case of `Collections.synchronizedMap(...)` a separate object (not the backing map) is used for synchronization. But essentially both `HashTable` and `Collections.synchronizedMap(...)` have similar

synchronization behavior. Note that a synchronized map backed by a `HashTable` simply adds a redundant synchronization layer and should never be used.

```
Collections.synchronizedMap(new Hashtable<>());
```

Finally, the `ConcurrentHashMap` uses sophisticated techniques to reduce the need for synchronization when multiple threads access the map in parallel. It is highly scalable and concurrent and can be iterated upon without requiring synchronization.

Property	HashMap	Hashtable	Collections.synchronizedMap(...)	ConcurrentHashMap
Null values/keys	yes	no	depends on backing map	no
Thread-safe	no	yes	yes	yes
Lock Mechanism	not applicable	locks the entire map	locks the entire map	locks a segment of the map
Iterator	fail-fast	fail-fast	fail-fast	weakly consistent

Performance#

We can run a simple and unsophisticated test to observe the performance of the different types of maps. The program in the widget below has 5 threads writing the same keys to a map with an initial capacity of 10. We time the run for each of the maps we discussed. The output demonstrates `ConcurrentHashMap` outperforms the other two maps and has a higher write throughput. Note, that this test is crude and doesn't take read performance into account but in general, the `ConcurrentHashMap` is the right choice in high concurrency environments.

```
import java.util.Collections;
import java.util.HashMap;
import java.util.Hashtable;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
```

```

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

        // start executor service
        ExecutorService es = Executors.newFixedThreadPool(5);

        performanceTest(new Hashtable<>(10), "Hashtable", es);
        performanceTest(Collections.synchronizedMap(new HashMap<>(10)),
        "Collections.synchronized(HashMap)", es);
        performanceTest(new ConcurrentHashMap<>(10), "Concurrent Hash Map", es);

        // shutdown the executor service
        es.shutdown();

    }

    static void performanceTest(Map<String, Integer> map, String mapName, ExecutorService es)
    throws Exception {
        Runnable task = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 1000000; i++)
                    map.put("key-" + i, i);
            }
        };
        long start = System.currentTimeMillis();

        Future future1 = es.submit(task);
        Future future2 = es.submit(task);
        Future future3 = es.submit(task);
        Future future4 = es.submit(task);
        Future future5 = es.submit(task);

        // wait for the threads to finish
        future1.get();
        future2.get();
        future3.get();
        future4.get();
        future5.get();

        long end = System.currentTimeMillis();

        System.out.println("Milliseconds taken using " + mapName + ":" + (end - start));
    }
}

```

In some runs you may observe **ConcurrentHashMap** taking longer than other maps for the writes to complete. This is because the code executes in the cloud

where execution environment (e.g. vCPUs, JVM version, compile-time optimization flags, memory etc) are beyond our control, however, if you ran the above program on a machine with multiple CPUs, the `ConcurrentHashMap` would outperform the other maps when executing majority of workloads.

Consider the following snippet:

```
Map map = Collections.synchronizedMap(new ConcurrentHashMap<>());
```

Which of the following statements is true?

A)

Snippet throws an exception because `ConcurrentHashMap` is already thread-safe.

B)

`map` variable is an instance of the `ConcurrentHashMap` class.

C)

`map` variable adds a layer of synchronization on the backing `ConcurrentHashMap`, effectively serializing access to the backing map one thread at a time and nullifying its ability to allow multiple writer threads to operate on it at the same time

ConcurrentModificationException

This lesson explains why `ConcurrentModificationException` occurs and how it can be avoided.

Single Thread Environment#

The name `ConcurrentModificationException` may sound related to *concurrency*, however, the exception can be thrown while a single thread operates on a map. In fact, `ConcurrentModificationException` isn't even part of the `java.util.concurrent` package. The exception occurs when a map is modified at the same time (concurrently) any of its collection views (keys, values or entry pairs) is being traversed. The program below demonstrates the exception being thrown as the main thread traverses the map entries and also attempts to insert new entries.

```
import java.util.*;

class Demonstration {
    public static void main( String args[] ) {
        HashMap<String, Integer> map = new HashMap<>();
```

```

// Fill the HashMap with some data
int i = 0;
for (i = 0; i < 100; i++) {
    map.put("key-" + i, i);
}

// Get an iterator for the entries in the map
Iterator it = map.entrySet().iterator();

while (it.hasNext()) {
    // Add a new key/value pair while the map is
    // being traversed.
    map.put("key-" + i, i);
    it.next();
    i++;
}
}
}

```

Multithread Environment#

In case of a single threaded environment it is often trivial to diagnose `ConcurrentModificationException` cause, however, in multithreaded scenarios, it may be difficult to do so as the exception may occur intermittently depending on how threads are scheduled for execution. Concurrent modification occurs when one thread is iterating over a map while another thread attempts to modify the map at the same time. A usual sequence of events is as follows:

1. Thread A obtains an iterator for the keys, values or entry set of a map.
2. Thread A begins to iterate in a loop.
3. Thread B comes along and attempts to delete, insert or update a key/value pair in the map.
4. `ConcurrentModificationException` is thrown when thread A attempts to retrieve the next item in the collection it is iterating.

Since the map has been modified from the time the iterator for the map was created, the thread iterating over the collection can observe inconsistent data and a `ConcurrentModificationException` is thrown. The program below demonstrates interaction between two threads that results in the exception.

```

import java.util.HashMap;
import java.util.Map;

```

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {

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

        HashMap<String, Integer> map = new HashMap<>();
        ExecutorService es = Executors.newFixedThreadPool(5);

        try {
            // create a task that slowly reads from the map.
            Runnable reader = new Runnable() {
                @Override
                public void run() {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException ie) { /*ignore*/ }

                    for (Map.Entry<String, Integer> entry : map.entrySet()) {
                        try {
                            Thread.sleep(1000);
                        } catch (InterruptedException ie) { /*ignore*/ }
                        System.out.println("key " + entry.getKey() + " value " + entry.getValue());
                    }
                }
            };
            // create a task to write to the map a little faster than the reader
            Runnable writer = new Runnable() {
                @Override
                public void run() {
                    for (int i = 0; i < 100; i++) {
                        try {
                            Thread.sleep(10);
                        } catch (InterruptedException ie) { /*ignore*/ }
                        map.put("key-" + i, i);
                    }
                }
            };
            // submit the task twice
            Future future1 = es.submit(writer);
            Future future2 = es.submit(reader);

            // wait for the threads to finish
        }
    }
}

```

```

        future1.get();
        future2.get();
    } finally {
        // We know the code will generate a ConcurrentModificationException so
        // remember to shutdown the executor service in a finally block
        es.shutdown();
    }

}
}

```

It is not only the `HashMap` that suffers from `ConcurrentModificationException`, other maps exhibit same behavior. The only map that is designed to be concurrently modified while being traversed is the `ConcurrentHashMap`. The program below demonstrates the behavior of all the maps.

```

import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

class Demonstration {
    public static void main( String args[] ) {
        test(new Hashtable<String, Integer>());
        test(new HashMap<String, Integer>());
        test(Collections.synchronizedMap(new HashMap<String, Integer>()));
        test(new ConcurrentHashMap<String, Integer>());
    }
}

static void test(Map<String, Integer> map) {

    // Put some data in the map
    int i;
    for (i = 0; i < 10; i++) {
        map.put("key-" + i, i);
    }

    Iterator it = map.entrySet().iterator();

    while (it.hasNext()) {
        map.put("key-" + i, i);
        try {
            it.next();
        } catch (ConcurrentModificationException ex) {
            System.out.println("ConcurrentModificationException thrown for map " +
map.getClass().getName());
            return;
        }
        i++;
    }
}

```

```

        System.out.println("No exception thrown for map " + map.getClass().getName());
    }
}

```

Even though the **ConcurrentHashMap** can undergo concurrent modifications (additions, deletions, updates) at the same time as its elements are being traversed, the modifications may not be reflected during the traversal. Consider the program below in which a reader thread starts traversing a map's entries while the map is being written to by a writer thread. The reader thread only observes a limited number of entries reflecting the state of the map at some point at or since the creation of the iterator/enumeration.

```

import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

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

        ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
        ExecutorService es = Executors.newFixedThreadPool(5);

        try {
            // create a reader that slowly reads from the map.
            Runnable reader = new Runnable() {
                @Override
                public void run() {

                    // wait for writer to put in some entries in the map
                    try {
                        Thread.sleep(20);
                    } catch (InterruptedException ie) { /*ignore*/ }

                    int seen = 0;
                    for (Map.Entry<String, Integer> entry : map.entrySet()) {
                        try {
                            Thread.sleep(10);
                        } catch (InterruptedException ie) { /*ignore*/ }
                        entry.getValue();
                        seen++;
                    }

                    System.out.println("Number of entries seen by reader thread : " + seen);
                }
            };
        };

        // create a writer that inputs 1000 entries in the map
    }
}

```

```

Runnable writer = new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            try {
                Thread.sleep(5);
            } catch (InterruptedException ie) { /*ignore*/ }
            map.put("key-" + i, i);
        }
        System.out.println("Writer thread finished.");
    }
};

// submit the two tasks
Future future1 = es.submit(writer);
Future future2 = es.submit(reader);

// wait for the threads to finish
future1.get();
future2.get();
} finally {
    es.shutdown();
}
}
}
}

```

In the above program, the writer inserts 1000 entries into the map but the reader only sees a handful.

As a user of `ConcurrentHashMap` one has to be cognizant of the limitation of iterators/enumerators, which may return a snapshot of the map taken at the time of creation of the iterator/enumeration or later.

```

static void quiz() {

    Map<String, Integer> map = new HashMap<>();
    Random random = new Random(System.currentTimeMillis());

    // Put some data in the map
    for (int i = 0; i < 10; i++) {
        map.put("key-" + i, i);
    }

    Iterator it = map.entrySet().iterator();

    while (it.hasNext()) {
        it.next();
        int k = random.nextInt(10);
        map.put("key-" + k, k);
    }
}

```

A)

Program throws `ConcurrentModificationException`

B)

Program doesn't throw any exception

B is correct!

The above scenario is very interesting since we are modifying the map but we are essentially overwriting the same key/value pair and no exception is thrown. In some cases such as handling duplicates (e.g. key/value pairs received from a message bus), a program can overwrite the same key/value pair twice (an example of *idempotent write*) and continue to function correctly but under different conditions may throw `ConcurrentModificationException`.

The program from the quiz is reproduced in the widget below.

```
import java.util.*;  
  
class Demonstration {  
    public static void main( String args[] ) {  
        Map<String, Integer> map = new HashMap<>();  
        Random random = new Random(System.currentTimeMillis());  
  
        // Put some data in the map  
        for (int i = 0; i < 10; i++) {  
            map.put("key-" + i, i);  
        }  
  
        Iterator it = map.entrySet().iterator();  
  
        while (it.hasNext()) {  
            it.next();  
            int k = random.nextInt(10);  
            map.put("key-" + k, k);  
        }  
  
        System.out.println("Program completes successfully.");  
    }  
}
```

Lock Interface

The Lock interface explained with examples.

Explanation#

The `Lock` interface provides a tool for implementing mutual exclusion that is more flexible and capable than `synchronized` methods and statements. A single thread is allowed to acquire the lock and gain access to a shared resource, however, some implementing classes such as the `ReentrantReadWriteLock` allow multiple threads concurrent access to shared resource. The use of `synchronized` methods or statements provides access to the implicit monitor lock associated with every object, but requires all lock acquisitions and releases to proceed in a block-structured way. Locks acquired in a nested fashion must be released in the exact opposite order, and all locks must be released in the same lexical scope in which they were acquired. These requirements restrict how `synchronized` methods and statements can be used and `Lock` implementations can be used for more complicated use-cases.

The `Lock` interface has the following classes implementing it:

1. `ReentrantLock`
2. `ReentrantReadWriteLock.ReadLock`
3. `ReentrantReadWriteLock.WriteLock`

Difference between Lock and Synchronized#

If you have worked with `synchronized` you may be wondering why we need the `Lock` interface and its implementing classes. The answer - locks offer additional functionality and far more flexibility in usage than synchronized methods and statements. For instance:

- A `Lock` can be tested for acquisition in a non-blocking fashion using the `trylock()` method
- A `Lock` can be waited upon for acquisition with a specified timeout using the `trylock(timeout)` method. After the timeout the thread abandons its attempt to acquire the lock and moves-on.
- A `Lock` can be waited upon for acquisition with the option to interrupt the acquiring thread using the `lockInterruptibly` method.
- Some `Lock` implementations also provide monitoring and deadlock detection. Additionally, `Lock` implementation can provide fair-use mode for locks, guaranteed ordering and non-reentrant use.

The flexibility and functionality of `Lock` implementations come at the cost of higher chance of human error since the locks are not automatically released as is the case with `synchronized` blocks and statements. The developer must remember to unlock the `Lock` in a `finally` block and as many times as the lock has been acquired for in case reentrancy is supported. The idiomatic use of any `Lock` implementation follows the below pattern:

```
Lock ourLock = // ... instantiate a lock  
  
ourLock.lock();  
try {  
    // ... Perform operations  
} finally {  
    ourLock.unlock();  
}
```

Using `Lock` and `synchronized`#

Finally, since `Lock` implementation is also an object it can be used as the argument to `synchronized` statement but such use is discouraged other than for internal implementation of the class. For example, the following is a bad practice:

```
Lock ourLock = // ... instantiate a lock  
  
synchronized(ourLock){  
    // ... Not a good idea  
}
```

LockSupport

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

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()` 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.

```
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");  
  
    }  
}
```

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.

```
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() {

                System.out.println(Thread.currentThread().getName() + " about to park.");
                try {
                    cyclicBarrier.await();
                } catch (InterruptedException | BrokenBarrierException e ){
                    // ignore
                }
                LockSupport.park();
                System.out.println(Thread.currentThread().getName() + " unparked.");
                try {
                    cyclicBarrier.await();
                } catch (InterruptedException | BrokenBarrierException e ){
                    // ignore
                }
            }
        });
        childThread.start();

        // wait for the childThread to start
        cyclicBarrier.await();
        System.out.println("Main thread about to unpark " + childThread.getName());
        LockSupport.unpark(childThread);
        cyclicBarrier.await();
        System.out.println("Main thread exiting");
    }
}
```

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.

Main.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() {
                    @Override
                    public void run() {
                        for (int i = 0; i < 10; i++) {
                            fifoLock.lock();
                            System.out.println(Thread.currentThread().getName() + " acquires the lock.");
                            // simulate work by sleeping
                            try {
                                Thread.sleep(ThreadLocalRandom.current().nextInt(20));
                            } catch (InterruptedException ie) {
                                // ignore for now
                            }
                            fifoLock.unlock();
                        }
                    }
                });
            }
        }
    }
}
```

```

        }
    });
}
} finally {
    executorService.shutdown();
    executorService.awaitTermination(1, TimeUnit.HOURS);
}

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

```

FIFOLock.java

```

import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.locks.LockSupport;

public class FIFOLock{

    private ConcurrentLinkedQueue<Thread> q = new ConcurrentLinkedQueue<>();
    private AtomicBoolean locked = new AtomicBoolean(false);

    public void lock() {

        // enqueue the current thread in the list of threads waiting
        // to acquire the lock
        q.add(Thread.currentThread());

        // we'll use compare and set method to change the status of the
        // FIFOLock to locked. Also, if the thread wakes up from the park
        // call we check for the predicate, that is the lock is available
        // for acquiring and also whether the woken-up thread is at the
        // head of the queue. Note, that park() method can experience
        // spurious wake-ups.
        while (q.peek() != Thread.currentThread()
            || !locked.compareAndSet(false, true)) {
            LockSupport.park();
        }

        // remove the head of the queue
        q.remove();
    }

    public void unlock() {

        // set the lock to false

```

```

locked.set(false);

// unpark the head of the queue
LockSupport.unpark(q.peek());

}
}

```

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.

Main.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 FIFOLock instance so that the spawned
        // thread parks itself when invoking lock()
        fifoLock.lock();

        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + " about to park itself.");
                fifoLock.lock();
            }
        });
    }
}

```

```

thread.start();

// wait for the child thread to get blocked
Thread.sleep(2000);

// now retrieve the blocker object associated with the thread

Object blocker = LockSupport.getBlocker(thread);
System.out.println(blocker);

// unlock so spawned thread can make progress
fifoLock.unlock();
}
}

```

FIFOLock.java

```

import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.locks.LockSupport;

public class FIFOLock{

    private ConcurrentLinkedQueue<Thread> q = new ConcurrentLinkedQueue<>();
    private AtomicBoolean locked = new AtomicBoolean(false);

    public void lock() {

        // enqueue the current thread in the list of threads waiting
        // to acquire the lock
        q.add(Thread.currentThread());

        // we'll use compare and set method to change the status of the
        // FIFOLock to locked. Also, if the thread wakes up from the park
        // call we check for the predicate, that is the lock is available
        // for acquiring and also whether the woken-up thread is at the
        // head of the queue. Note, that park() method can experience
        // spurious wake-ups.
        while (q.peek() != Thread.currentThread()
            || !locked.compareAndSet(false, true)) {
            // we pass the lock object itself to the park() method
            LockSupport.park(this);
        }
        // remove the head of the queue
        q.remove();
    }
}

```

```

public void unlock() {

    // set the lock to false
    locked.set(false);

    // unpark the head of the queue
    LockSupport.unpark(q.peek());

}

}

```

ReentrantLock

This lesson explains usage of the ReentrantLock.

The `ReentrantLock` implements the `Lock` interface and is functionally similar to the implicit monitor lock accessed using `synchronized` methods and statements.

The lock is said to be owned by the thread that `locks` it and any other thread attempting to `lock` the object will block. A thread that already owns the lock will return immediately if it invokes `lock` again. The reentrant behavior of the lock allows recursively locking by the already owning thread, however, the lock supports a maximum of 2147483647 locks by the same thread.

Idiomatic Use of Lock#

Threads can experience deadlocks when locks aren't unlocked after use. The correct idiomatic usage of a lock should follow the below pattern:

```

lock.lock();  // acquire the lock
try {
    // ... functionality to be executed in the method body
} finally {
    lock.unlock() // must release the lock in a finally block
}

```

If you acquire a lock and then continue execution without a `try` block it is possible that an exception occurs and the lock is never released even though the program continues to execute. Depending on how the program is structured it is possible that the program experiences a deadline as it progresses.

Fairness#

The `ReentrantLock` can also be operated in *fair mode* where the lock is granted to the longest waiting thread. Thus no thread experiences starvation and the variance in times to obtain the lock is also small. Without the *fair mode* the lock doesn't guarantee the order in which threads acquire the lock. When a lock is operated in fair mode in an environment with several threads contending access to the lock, throughput suffers and is significantly reduced. Finally, the method `tryLock` when invoked without timeout doesn't honor the fairness setting and acquires the lock if it is free even in the presence of other waiting threads.

Example#

Consider the example below which has three threads, the main thread, `threadA` and `threadB` interacting with an instance of `ReentrantLock`. The main thread locks the lock object thrice and invokes `unlock` on the object with an artificially introduced delay. During this time, `threadA` does a busy spinning waiting for the lock to be free. `threadA` uses the method `tryLock()` to check if it can acquire the lock. On the other hand `threadB` simply acquires and then releases the lock. `threadB` will inevitably block at acquiring the lock and the method `getQueueLength()` will display a count of 1. The example also demonstrates the use of the method `getHoldCount()` which either returns 0 if the lock isn't held by the thread invoking the method or the number of times the owning thread has recursively acquired the lock. Pay attention to how we use this method in the `finally` block of the main thread to `unlock` as many times as required if an exception occurs.

Finally, there's the `isLocked()` method that returns true if the lock is held by any thread. However, note that both methods `isLocked()` and `getQueueLength()` are designed for monitoring the state of the system and shouldn't be used in program control, e.g. you should never do something like below:

```
if (lock.isLocked()) {
    // Take some action
}
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

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

```

ExecutorService es = Executors.newFixedThreadPool(5);
ReentrantLock lock = new ReentrantLock();
Runnable threadA = new Runnable() {
    @Override
    public void run() {
        threadA(lock);
    }
};

Runnable threadB = new Runnable() {
    @Override
    public void run() {
        threadB(lock);
    }
};

try {
    lock.lock();
    lock.lock();
    lock.lock();
    lock.lock();

    System.out.println("Main thread lock hold count = " + lock.getHoldCount());

    // submit other threads
    Future future1 = es.submit(threadA);
    Future future2 = es.submit(threadB);

    // release locks slowly
    for (int i = 0; i < 3; i++) {
        Thread.sleep(50);
        lock.unlock();
    }

    System.out.println("Main thread released lock. Lock hold count = " +
lock.getHoldCount());
    future1.get();
    future2.get();
} finally {
    // Make sure to release the locks if an exception occurs
    for (int i = 0; i < lock.getHoldCount(); i++) {
        lock.unlock();
    }

    // Shutdown the executor service
    es.shutdown();
}
}

```

```

static void threadB(Lock lock) {
    lock.lock();
    lock.unlock();
}

static void threadA(ReentrantLock lock) {

    String name = "THREAD-A";
    Thread.currentThread().setName(name);
    boolean keepTrying = true;

    System.out.println("Is lock owned by any other thread = " + lock.isLocked());

    while (keepTrying) {
        System.out.println(name + " trying to acquire lock");

        if (lock.tryLock()) {
            try {
                System.out.println(name + " acquired lock");
                keepTrying = false;
            } finally {
                lock.unlock();
                System.out.println(name + " released lock");
            }
        } else {
            System.out.println(name + " failed to acquire lock. Other threads waiting = " +
lock.getQueueLength());
        }
    }

    try {
        Thread.sleep(20);
    } catch (InterruptedException ie) {
        // ignore exception.
    }
}
}

```

```

public class question {

    static ReentrantLock lock = new ReentrantLock();

    static int test() {

        lock.lock();

        try {
            return -1;
        } finally {

```

```
        lock.unlock();
        System.out.println("Unlocked");
    }

    System.out.println("Exiting Program");
}
}
```

A)

“Unlocked” will be printed but “Exiting Program” is unreachable statement.

B)

Both print statements are unreachable.

C)

Both print statements are executed.

D)

`lock` object is never unlocked because of the `return` statement.

ReadWriteLock

This lesson examines the `ReadWriteLock` interface and the implementing class `ReentrantReadWriteLock`. The lock is intended to allow multiple readers to read at a time but only allow a single writer to write.

The `ReadWriteLock` interface is part of Java’s `java.util.concurrent.locks` package. The only implementing class for the interface is `ReentrantReadWriteLock`. The `ReentrantReadWriteLock` can be locked by multiple readers at the same time while writer threads have to wait. Conversely, the `ReentrantReadWriteLock` can be locked by a single writer thread at a time and other writer or reader threads have to wait for the lock to be free.

ReentrantReadWriteLock#

The `ReentrantReadWriteLock` as the name implies allows threads to recursively acquire the lock. Internally, there are two locks to guard for read and write accesses. `ReentrantReadWriteLock` can help improve concurrency over using a mutual exclusion lock as it allows multiple reader threads to read concurrently. However, whether an application will truly realize concurrency improvements depends on other factors such as:

- Running on multiprocessor machines.

- Frequency of reads and writes. Generally, `ReadWriteLock` can improve concurrency in scenarios where read operations occur frequently and write operations are infrequent. If write operations happen often then most of the time is spent with the lock acting as a mutual exclusion lock.
- Contention for data, i.e. the number of threads that try to read or write at the same time.
- Duration of the read and write operations. If read operations are very short then the overhead of locking `ReadWriteLock` versus a mutual exclusion lock can be higher.

In practice, you'll need to evaluate the access patterns to the shared data in your application to determine the suitability of using the `ReadWriteLock`.

Fair Mode#

The `ReentrantReadWriteLock` can also be operated in the *fair mode*, which grants entry to threads in an approximate arrival order. The longest waiting writer thread or a group of longest waiting reader threads is given preference to acquire the lock when it becomes free. In case of reader threads we consider a group since multiple reader threads can acquire the lock concurrently.

Cache Example#

One common scenario where there can be multiple readers and writers is that of a cache. A cache is usually used to speed up read requests from another data source e.g. data from hard disk is cached in memory so that a request doesn't have to wait for data to be fetched from the hard disk thus saving I/O. Usually, there are multiple readers trying to read from the cache and it is imperative that the readers don't step over writers or vice versa.

In the simple case we can relax the condition that readers are ok to read stale data from the cache. We can imagine that a single writer thread periodically writes to the cache and readers don't mind if the data gets stale before the next update by the writer thread. In this scenario, the only caution to exercise is to make sure no readers are reading the cache when a writer is in the process of writing to the cache.

In the example program below we use a `HashMap` as a cache and input a single key/value pair that is periodically updated by the writer thread. From

the output of the program, note that the reader threads find the value of the key updated after the acquisition of the lock by the writer thread.

```
import java.util.HashMap;
import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

class Demonstration {

    static Random random = new Random();

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

        ExecutorService es = Executors.newFixedThreadPool(15);

        // cache
        HashMap<String, Object> cache = new HashMap<>();
        ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

        // put some data in the cache
        cache.put("key", -1);

        Runnable writerTask = new Runnable() {
            @Override
            public void run() {
                writerThread(cache, lock);
            }
        };

        Runnable readerTask = new Runnable() {
            @Override
            public void run() {
                readerThread(cache, lock);
            }
        };

        try {
            // submit tasks for execution
            Future future1 = es.submit(writerTask);
            Future future2 = es.submit(readerTask);
            Future future3 = es.submit(readerTask);
            Future future4 = es.submit(readerTask);
            Future future5 = es.submit(readerTask);
```

```

        // wait for tasks to finish
        future1.get();
        future2.get();
        future3.get();
        future4.get();
        future5.get();
    } finally {
        es.shutdown();
    }

}

static void writerThread(HashMap<String, Object> cache, ReadWriteLock lock) {

    for (int i = 0; i < 9; i++) {
        try {
            Thread.sleep(random.nextInt(50));
        } catch (InterruptedException ie) {
            // ignore
        }
        lock.writeLock().lock();
        try {
            System.out.println("Acquired write lock");
            cache.put("key", random.nextInt(1000));
        } finally {
            lock.writeLock().unlock();
        }
    }
}

static void readerThread(HashMap<String, Object> cache, ReadWriteLock lock) {

    for (int i = 0; i < 3; i++) {
        try {
            Thread.sleep(random.nextInt(100));
        } catch (InterruptedException ie) {
            // ignore
        }
        lock.readLock().lock();
        try {
            System.out.println("Acquire read lock and reading key = " + cache.get("key"));
        } finally {
            lock.readLock().unlock();
        }
    }
}

```

```
    }  
}
```

Downgrading to Read Lock#

To showcase downgrading from a write lock to a read lock, we'll slightly modify our scenario to include a challenging requirement that readers can't tolerate stale data. We'll assume that readers have the ability to trigger an update of the cache data if it is found to be stale. A boolean flag `isDataFresh` depicts whether data in the cache is fresh or not. We'll make changes to our previous program to accommodate for these new requirements. The writer thread will now occasionally set the flag `isDataFresh` to false to indicate that the reader threads must trigger an update. The writer thread from our previous program doesn't write data to the cache anymore rather it simply sets the flag `isDataFresh` to false to force the reader threads to trigger an update.

The astute reader will realize this setup implies that a reader thread must acquire the write lock upon discovering the data is stale and initiate an update. Thus the reader thread works with both the read and the write locks. The reader thread upon finding the data is stale acquires the write lock, triggers a cache refresh, and then downgrades to the read lock to continue reading data. Note that the vice versa, i.e. upgrading a read lock to a write lock isn't allowed.

The comments in the program below explain the program flow and the downgrading of write lock.

```
import java.util.HashMap;  
import java.util.Random;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
import java.util.concurrent.Future;  
import java.util.concurrent.locks.ReadWriteLock;  
import java.util.concurrent.locks.ReentrantReadWriteLock;  
  
class Demonstration {  
  
    static Random random = new Random();  
    static boolean isDataFresh = true;  
  
    public static void main( String args[] ) throws Exception {  
        ExecutorService es = Executors.newFixedThreadPool(15);  
  
        // cache  
        HashMap<String, Object> cache = new HashMap<>();  
        ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
```

```

// put some data in the cache
cache.put("key", -1);

Runnable writerTask = new Runnable() {
    @Override
    public void run() {
        writerThread(lock);
    }
};

Runnable readerTask = new Runnable() {
    @Override
    public void run() {
        readerThread(cache, lock);
    }
};

try {
    Future future1 = es.submit(writerTask);
    Future future2 = es.submit(readerTask);
    Future future3 = es.submit(readerTask);
    Future future4 = es.submit(readerTask);

    future1.get();
    future2.get();
    future3.get();
    future4.get();
} finally {
    es.shutdown();
}
}

static void writerThread(ReadWriteLock lock) {

    for (int i = 0; i < 9; i++) {
        try {
            Thread.sleep(random.nextInt(50));
        } catch (InterruptedException ie) {
            // ignore
        }

        lock.writeLock().lock();
        System.out.println("Acquired write lock");
        isDataFresh = false;
        lock.writeLock().unlock();
    }
}

```

```

static void updateData(HashMap<String, Object> cache) {
    cache.put("key", random.nextInt(1000));
    isDataFresh = true;
}

static void readerThread(HashMap<String, Object> cache, ReadWriteLock lock) {

    for (int i = 0; i < 3; i++) {
        try {
            Thread.sleep(random.nextInt(50));
        } catch (InterruptedException ie) {
            // ignore
        }

        // acquire the read lock to check if data is fresh before
        // reading from the cache
        lock.readLock().lock();

        try {
            // check if the data is fresh
            if (!isDataFresh) {

                // release the read lock, before acquiring the write lock
                lock.readLock().unlock();

                // acquire the write lock before triggering an update
                lock.writeLock().lock();

                try {
                    // Check the flag again, the data might already have been refreshed by
                    // another writer thread.
                    if (!isDataFresh) {
                        updateData(cache);
                    }

                    // acquire read lock before releasing the write lock. This is an
                    // example of downgrading from write -> read lock
                    lock.readLock().lock();
                } finally {
                    lock.writeLock().unlock();
                }
            }
        }
    }

    System.out.println("Acquire read lock and reading key = " + cache.get("key"));

} finally {

```

```
        lock.readLock().unlock();  
    }  
}  
}  
}  
}
```

The output of the above program is similar to the output of the previous program. Note, the acquisition of the write lock each time triggers a refresh of the `cache` as depicted by a change the key value read by the reader threads.

Reentrancy#

Finally, remember that the `ReentrantReadWriteLock` allows thread to recursively acquire the read or write lock. You must remember to unlock as many times as you lock. The sequence of locks in the snippet below will result in a deadlock since the write lock is acquired twice but released only once before attempting to acquire the read lock.

```
ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
lock.writeLock().lock();
lock.writeLock().lock();
lock.writeLock().unlock();
lock.readLock().lock();
```

Consider the code snippet below:

```
ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
lock.readLock().lock();
lock.writeLock().lock();
```

What will be the outcome when a thread executes this snippet:

A)

The thread will block indefinitely because read lock can't be upgraded to write lock.

B)

The thread will throw an exception because read lock can't be upgraded to a write lock.

C)

The setup is allowed and the thread will acquire both read and write locks.

StampedLock

Learn how to use the StampedLock class for optimistic read, write and read access.

We'll cover the following

- Overview
- Modes
 - Reading
 - Writing
 - Optimistic Read
- Converting modes
 - Upgrade to write lock after optimistic read
 - Upgrade to write lock from read lock
 - Downgrade to read lock from write lock
 - Downgrade to optimistic read from read lock
 - Failed upgrade to write lock example
- Characteristics
- Conclusion

Overview#

The `StampedLock` was introduced in Java 9 and is a capability-based lock with three modes for controlling read/write access. The `StampedLock` class is

designed for use as an internal utility in the development of other thread-safe components. Its use relies on knowledge of the internal properties of the data, objects, and methods it protects.

Modes#

The state of a `StampedLock` is defined by a version and mode. There are three modes the lock can be in:

- Writing
- Reading
- Optimistic Reading

On acquiring a lock, a stamp (long value) is returned that represents and controls access with respect to a lock state. The stamp can be used later on to release the lock or convert the existing acquired lock to a different mode.

Reading#

The read mode can be acquired using the `readLock()` method. When reading a thread isn't expected to make any changes to the shared state and therefore it makes sense to have multiple threads acquire the read lock at the same time. The method returns a stamp that can be used to unlock the read lock. Timed and untimed versions of `tryReadLock()` also exist.

The code widget below demonstrates multiple threads acquiring the `StampedLock` in read mode.

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

class Demonstration {
    public static void main( String args[] ) {
        ExecutorService executorService = Executors.newFixedThreadPool(10);

        // create an instance of StampedLock
        StampedLock stampedLock = new StampedLock();

        // main thread attempts to acquire the lock twice, which is granted
        long readStamp1 = stampedLock.readLock();
        long readStamp2 = stampedLock.readLock();

        try {
            // create 3 threads
            for (int i = 0; i < 3; i++) {
                executorService.submit(new Runnable() {
```

```

@Override
public void run() {
    try {
        long readStamp = stampedLock.readLock();
        System.out.println("Read lock count in spawned thread " +
stampedLock.getReadLockCount();

        // simulate thread performing read of shared state
        try {
            Thread.sleep(2000);
        } catch (InterruptedException ie) {
            // ignore
        } finally {
            stampedLock.unlockRead(readStamp);
        }
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}

// let the main thread simulate work for 5 seconds
Thread.sleep(5000);

} finally {
    // wait for spawned threads to finish
    executorService.shutdown();
    executorService.awaitTermination(1, TimeUnit.HOURS);

    // remember to unlock
    stampedLock.unlock(readStamp1);
    stampedLock.unlock(readStamp2);
}

System.out.println("Read lock count in main thread " + stampedLock.getReadLockCount());
System.out.println("stampedLock.isReadLocked() " + stampedLock.isReadLocked());
}
}

```

While read lock is held by a reader thread, all attempts to acquire the write lock will be blocked.

Writing#

The write mode can be acquired by invoking the `writeLock()` method. When the write lock is held, all threads attempting to acquire the read lock will be blocked. Similar to the read lock, timed and untimed versions of `tryWriteLock()` exist. The following widget demonstrates the use of the write lock:

```

import java.util.concurrent.*;
class Demonstration {

    public static void main( String args[] ) {
        ExecutorService executorService = Executors.newFixedThreadPool(10);

        // create an instance of StampedLock
        StampedLock stampedLock = new StampedLock();

        // main thread acquires the write lock before spawning read thread
        long stamp = stampedLock.writeLock();

        try {
            executorService.submit(new Runnable() {
                @Override
                public void run() {
                    System.out.println("Attempting to acquire read lock");
                    long readStamp = stampedLock.readLock();
                    System.out.println("Read lock acquired");
                    stampedLock.unlock(readStamp);
                }
            });
        }

        // Having the thread sleep for 3 seconds so that the read thread
        // gets blocked
        Thread.sleep(3000);

        } finally {
            // unlock write
            System.out.println("Write lock being released");
            stampedLock.unlock(stamp);

            // wait for read thread to exit
            executorService.shutdown();
            executorService.awaitTermination(1, TimeUnit.HOURS);
        }
    }
}

```

In the above program, the print statement for releasing the write lock is always output before the statement for acquiring the read lock.

Remember that the `writeLock()` is not re-entrant. The following program results in a deadlock:

```

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

```

```

// create an instance of StampedLock
StampedLock stampedLock = new StampedLock();

stampedLock.writeLock();

stampedLock.writeLock();
}
}

```

Optimistic Read#

Consider a scenario, where we never want to block a thread from acquiring the write lock if the lock isn't already acquired. Recall, that if the read lock is already held then a thread attempting to acquire the write lock will block. `StampedLock` makes this scenario possible with the optimistic reading mode. A thread can invoke the `tryOptimisticRead()` method which returns a non-zero value if the write lock isn't exclusively locked. The thread can then proceed to read but once the thread reads the desired value it must validate if the lock wasn't acquired for a write in the meanwhile. If validation returns true then the reader thread can safely assume that from the time it was returned a stamp from invoking the `tryOptimisticRead()` method till the time the thread validated the stamp, the lock hasn't been acquired for writing and the data hasn't changed. In case, if the lock was acquired for a write in the intervening period, then the validation will return false. The validation can be performed using the method `validate(long)` which takes in the stamp issued at the time of invoking `tryOptimisticRead()`.

This mode can be broken at any time by a thread acquiring a write lock. Another way to think of this mode is as an extremely weak version of a read-lock. Using optimistic reads in cases where the read happens briefly can improve throughput and reduce contention.

Note that `tryOptimisticRead()` doesn't acquire the read lock nor is the read lock count incremented as the following snippet demonstrates:

```

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

        ExecutorService executorService = Executors.newFixedThreadPool(10);

        // create an instance of StampedLock
        StampedLock stampedLock = new StampedLock();

        // optimistic read
        stampedLock.tryOptimisticRead();

        // outputs "read lock count 0 is read locked false"
    }
}

```

```

        System.out.println("read lock count " + stampedLock.getReadLockCount() + " is read locked
" + stampedLock.isReadLocked());
    }
}

```

The sequence of operations in the following code widget illustrates when a stamp returned from a `tryOptimisticRead()` call becomes invalid:

```

class Demonstration {
    public static void main( String args[] ) {
        // create an instance of StampedLock
        StampedLock stampedLock = new StampedLock();

        // try optimistic read
        long stamp = stampedLock.tryOptimisticRead();

        // check for validation, prints true
        System.out.println(stampedLock.validate(stamp));

        // acquire the write lock which will invalidate the previous stamp
        stampedLock.writeLock();

        // check for validation, prints false
        System.out.println(stampedLock.validate(stamp));
    }
}

```

In the above program, instead of the write lock if we acquire the read lock, the validation will come out to be true. The official Java documentation marks optimistic reads as inherently fragile and cautions “*optimistic read sections should only read fields and hold them in local variables for later use after validation. Fields read while in optimistic mode may be wildly inconsistent, so usage applies only when you are familiar enough with data representations to check consistency and/or repeatedly invoke method validate(). For example, such steps are typically required when first reading an object or array reference, and then accessing one of its fields, elements or methods.*”

The following program demonstrates the idiomatic use of optimistic read. Consider a function that is always passed-in an integer array with three elements. The function computes the product of the three numbers and returns the answer. We'll also assume that the multiplication doesn't result in an overflow exception. The important detail is to remember to read in the array elements into local variables, validate the stamp and then compute the product using the local variables. The reason to store values in local variables is because just after stamp validation, we can't be sure if the original passed-in array remains unmodified. With storing the array values in local variables, we have in a sense, preserved a snapshot of the state of the data, i.e. the array and computed a product for that snapshot. In case

the stamp validation fails, then we are compelled to acquire the read lock and compute the product. The code with comments explaining the program flow appears below:

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

class Demonstration {

    static StampedLock stampedLock = new StampedLock();

    public static void main( String args[] ) {
        int[] array = new int[3];
        array[0] = 3;
        array[1] = 5;
        array[2] = 7;

        productOfThree(array);
    }

    static int productOfThree(int[] array) {

        // get a stamp from optimistic read
        long stamp = stampedLock.tryOptimisticRead();

        // read the three elements of the array in local variables
        int num1 = array[0];
        int num2 = array[1];
        int num3 = array[2];

        // if stamp isn't valid anymore i.e. a write lock was acquired then
        // get the read lock

        if (!stampedLock.validate(stamp)) {

            // this call may block
            stamp = stampedLock.readLock();

            try {
                return array[0] * array[1] * array[3];
            } catch (RuntimeException re) {
                // log exception
                throw re;
            } finally {
                // remember to unlock in finally block
                stampedLock.unlockRead(stamp);
            }
        }
    }
}
```

```

    // assuming the multiplication doesn't result in an overflow exception
    return num1 * num2 * num3;
}
}

```

Coverting modes#

The `StampedLock` also provides support for converting locks from one mode to another if certain conditions are met. For instance, the method `tryConvertToWriteLock(long)` upgrades to a write lock if one of the following is true:

- The write lock is currently held.
- The read lock is current held and no other readers exists
- The optimistic mode is in progress and the write lock is available

Here are some examples of upgrading and downgrading across modes:

Upgrade to write lock after optimistic read#

```

// create an instance of StampedLock
StampedLock stampedLock = new StampedLock();

// get into optimistic read mode
long stamp = stampedLock.tryOptimisticRead();

// upgrade to write lock
stampedLock.tryConvertToWriteLock(stamp);

/*
 * output of program is
 * Read locks held : 0
 * Write lock held : true
 */
System.out.println("Read locks held : " + stampedLock.getReadLockCount() + "\nWrite lock held : " + stampedLock.isWriteLocked());

```

Upgrade to write lock from read lock

```

// create an instance of StampedLock
StampedLock stampedLock = new StampedLock();

// acquire read lock
long stamp = stampedLock.readLock();

// upgrade to write lock
stampedLock.tryConvertToWriteLock(stamp);

/*
 * output of program is
 */

```

```

    * Read locks held : 0
    * Write lock held : true
    */
System.out.println("Read locks held : " + stampedLock.getReadLockCount() + "\nWrite lock held : " + stampedLock.isWriteLocked());

```

Downgrade to read lock from write lock#

```

// create an instance of StampedLock
StampedLock stampedLock = new StampedLock();

// acquire write lock
long stamp = stampedLock.writeLock();

// downgrade to read lock
stampedLock.tryConvertToReadLock(stamp);

/*
 * output of program is
 * Read locks held : 1
 * Write lock held : false
 */
System.out.println("Read locks held : " + stampedLock.getReadLockCount() + "\nWrite lock held : " + stampedLock.isWriteLocked());

```

Downgrade to optimistic read from read lock#

```

// create an instance of StampedLock
StampedLock stampedLock = new StampedLock();

// acquire read lock
long stamp = stampedLock.readLock();

// downgrade to optimistic read mode
stampedLock.tryConvertToOptimisticRead(stamp);

/*
 * output of program is
 * Read locks held : 0
 * Write lock held : false
 */
System.out.println("Read locks held : " + stampedLock.getReadLockCount() + "\nWrite lock held : " + stampedLock.isWriteLocked());

```

Failed upgrade to write lock example#

The final example demonstrates a failed attempt by the main thread to upgrade to the write lock from optimistic read mode. Another spawned thread holds the write lock and prevents the main thread from upgrading. The output of the program appears below:

```
Stamp : 0
Read locks held : 0
Write lock held : true
spawned thread exiting.
```

The code widget below includes comments that explain the failed lock upgrade example:

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

class Demonstration {

    public static void main( String args[] ) {
        ExecutorService executorService = Executors.newFixedThreadPool(10);

        // create an instance of StampedLock
        StampedLock stampedLock = new StampedLock();
        long stamp = stampedLock.tryOptimisticRead();

        try {
            executorService.submit(new Runnable() {
                @Override
                public void run() {
                    long stamp = stampedLock.writeLock();

                    // simulate work by sleeping
                    try {
                        Thread.sleep(6000);
                        System.out.println("spawned thread exiting.");
                    } catch (InterruptedException ie) {
                        // ignore
                    } finally {
                        stampedLock.unlockWrite(stamp);
                    }
                }
            });
        }

        // give the spawned thread a chance to acquire the write lock. NEVER use
        // Thread.sleep() for thread synchronization in production code!
        Thread.sleep(3000);

        // attempt to upgrade to write lock
        stamp = stampedLock.tryConvertToWriteLock(stamp);
        System.out.println("Stamp : " + stamp + "\nRead locks held : " +
stampedLock.getReadLockCount() + "\nWrite lock held : " + stampedLock.isWriteLocked());

    } finally {
        executorService.shutdown();
    }
}
```

```
        executorService.awaitTermination(1, TimeUnit.HOURS);  
    }  
}  
}
```

Characteristics#

Some of the salient notes/features about the `StampedLock` include:

- Similar to `Semaphore` the `StampedLock` has no notion of ownership. Lock acquired by one thread can be released by another thread, which isn't possible for implementation of the `Lock` interface.
 - The `StampedLock` class doesn't implement the `Lock` or `ReadWriteLock` interfaces but applications that desire to use the functionality offered by these interfaces out of an instance of `StampedLock` can do so using the methods `asReadLock()`, `asWriteLock()` and `asReadWriteLock()`. The following statements are possible

```
// create an instance
StampedLock stampedLock = new StampedLock();

// use functionality as a ReadLock
Lock readLock = stampedLock.asReadLock();

// use functionality as a WriteLock
Lock writeLock = stampedLock.asWriteLock();

// use functionality as a ReadWriteLock
ReadWriteLock readWriteLock = stampedLock.asReadWriteLock();
```

- The scheduling policy of `StampedLock` does not consistently prefer readers over writers or vice versa. All `try*` methods are best-effort and do not necessarily conform to any scheduling or fairness policy.
 - A zero return from any `try*` method for acquiring or converting locks does not imply or carry any information about the state of the lock; a subsequent invocation may succeed.
 - Stamp values can recycle after a year but not earlier. This implies that if a stamp is held without use or validation for longer than this period may fail to validate correctly.
 - The `StampedLock` class is serializable, but always deserializes into the initial unlocked state, so its instances are not useful for remote locking.

- Stamps aren't cryptographically secure and a valid stamp can be guessed by malicious participants in a system.
- The `StampedLock`'s write lock is not reentrant i.e. recursively attempting to acquire the lock will result in a deadlock as the following snippet demonstrates. The execution times out because the main thread deadlocks itself by acquiring the write lock twice.

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

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

        // create an instance
        StampedLock stampedLock = new StampedLock();

        stampedLock.writeLock();
        stampedLock.writeLock();

    }
}
```

- Don't use optimistic read locks without validation if the read sections aren't tolerant to potential inconsistencies.

Conclusion#

`StampedLock` is an advanced locking construct, which a typical Java developer is unlikely to encounter in her day to day work. `StampedLock` class offers flexible usage and functionality but requires careful thought in its application to avoid bugs that can be introduced in a program especially when using the optimistic read mode.

What is the outcome of running the below snippet:

```
void demoWriteLock() throws Exception {

    ExecutorService executorService = Executors.newFixedThreadPool(10)
;

    StampedLock stampedLock = new StampedLock();

    // main thread acquires the write lock before spawning read thread
    long stamp = stampedLock.writeLock();

    try {
        executorService.submit(new Runnable() {
            @Override
            public void run() {
                System.out.println("Attempting to acquire read lock");
            }
        });
    }
}
```

```

        long readStamp = stampedLock.readLock();
        System.out.println("Read lock acquired");
        stampedLock.unlock(readStamp);
    }
});

Thread.sleep(3000);

} finally {
    // wait for read thread to exit
    executorService.shutdown();
    executorService.awaitTermination(1, TimeUnit.HOURS);

    // unlock write
    System.out.println("Write lock being released");
    stampedLock.unlock(stamp);
}
}

```

A)

The program results in a deadlock

B)

The “Write lock being released” is printed after the “Read lock acquired”

C)

The “Read lock acquired” is printed after the “Write lock being released”

D)

Program throws an exception

Atomic Boolean

Get a clear understanding and use cases of the `AtomicBoolean` class and its differences with `volatile` boolean variables.

Explanation#

The `AtomicBoolean` class belonging to

Java’s `java.util.concurrent.atomic` package represents a boolean value that can be updated and modified atomically. The `Atomic*` family of classes extend the notion of `volatile` variables that are designed to be operated upon without locking using machine-level atomic instructions available on modern processors. However, on other platforms some form of internal locking may be used to serialize thread access.

`Atomic*` classes including `AtomicBoolean` offer a method `compareAndSet(expectedValue, updatedValue)` to conditionally update the value of the variable to `updatedValue` if it is set to `expectedValue` in one go, i.e. atomically. All read-and-update methods except for `lazySet()` and `weakCompareAndSet()` have memory effects equivalent of both reading and writing `volatile` variables.

The read and write methods i.e. `get()` and `set()` on instances of this class are similar in behavior to volatile variables i.e `get()` has the memory effect of reading a volatile variable and `set()` has the memory effect of writing a volatile variable.

Difference between `Volatile boolean` and `AtomicBoolean`#

It may be tempting to confuse and equate `volatile boolean` with `AtomicBoolean`, however, the two differ in several respects. Though it is fair to say that `volatile boolean` exhibits a subset of the functionality of `AtomicBoolean`. Recall, from our discussion on the `volatile` keyword that Java's memory model and compiler optimization can result in a situation where one thread doesn't see the latest value of a shared variable in main memory since it is using a cached stale value. Marking such a shared variable as `volatile` ensures that all threads see the latest state of the variable at all times. The quintessential example of this scenario found in online literature is as follows:

```
class SomeClass {  
    boolean quit = false;  
  
    void thread1() {  
        while (!quit) {  
            // ... Take some action  
        }  
    }  
  
    void thread2() {  
        // thread 1 may never see the updated value of quit!  
        quit = true;  
    }  
}
```

Without `volatile` the thread executing the method `thread1` in the above program can spin in an infinite loop and never quit. Apart from delivering a consistent view of the memory, the `volatile` keyword doesn't promise much in synchronization guarantees. Specifically, multiple threads accessing a `volatile` keyword don't do so in a serialized manner. The onus of making a

volatile variable's accesses synchronized and thread-safe is on the developer. This is where the `AtomicBoolean` classes come in. For instance, the methods `compareAndSet()` and `getAndSet()` exposed by the `AtomicBoolean` class represent a series of operations executed atomically, which otherwise would require synchronization on the part of the developer and are unachievable as an atomic transaction using `volatile` variables.

To make the distinction clearer let's work with an example. Say, we instantiate several threads and have them race to change a shared boolean variable's `won` value to true. The first one to do so is considered to have won the race. The rest print a message to indicate they lost the race. We mark the variable `won` as volatile and run the program below:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {

    static volatile boolean won = false;

    public static void main( String args[] ) throws Exception {
        ExecutorService es = Executors.newFixedThreadPool(25);
        try {
            int numThreads = 15;
            Runnable[] racers = new Runnable[numThreads];
            Future[] futures = new Future[numThreads];

            // create thread tasks
            for (int i = 0; i < numThreads; i++) {
                racers[i] = new Runnable() {
                    @Override
                    public void run() {
                        race();
                    }
                };
            }

            // submit threads for execution
            for (int i = 0; i < numThreads; i++) {
                futures[i] = es.submit(racers[i]);
            }

            // wait for threads to finish
            for (int i = 0; i < numThreads; i++) {
                futures[i].get();
            }
        } finally {
    }
}
```

```

        es.shutdown();
    }
}

static void race() {
    if (!won) {
        won = true;
        System.out.println(Thread.currentThread().getName() + " won the race.");
    } else {
        System.out.println(Thread.currentThread().getName() + " lost.");
    }
}

}

```

If you run the above program enough times especially on a machine with multiple processes you'll observe multiple threads printing the winning statement. The widget above may not show the faulty run since the VM executing the code may have a single vCPU. The `won` variable doesn't prevent multiple threads from accessing and then updating the variable `won`. A possible sequence resulting in multiple threads declaring themselves as winners can be:

1. Thread A reads the value of `won` which is false.
2. Thread B reads the value of `won` which is false.
3. Thread A changes the value of `won` to true and the variable is updated in main memory for all threads to see.
4. Thread B doesn't know Thread A had read the value of the variable `won` before Thread B accessed it. Thread B too updates the value of `won` to true.
5. Since `won` is a volatile variable, it always reflects its latest value to the next thread reading it but that's about it.

The above program can be fixed by using appropriate synchronization as shown in the widget below:

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {

    static Object syncObj = new Object();
    static boolean won = false;

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

```

```

ExecutorService es = Executors.newFixedThreadPool(25);
try {
    int numThreads = 15;
    Runnable[] racers = new Runnable[numThreads];
    Future[] futures = new Future[numThreads];

    // create thread tasks
    for (int i = 0; i < numThreads; i++) {
        racers[i] = new Runnable() {
            @Override
            public void run() {
                race();
            }
        };
    }

    // submit threads for execution
    for (int i = 0; i < numThreads; i++) {
        futures[i] = es.submit(racers[i]);
    }

    // wait for threads to finish
    for (int i = 0; i < numThreads; i++) {
        futures[i].get();
    }
} finally {
    es.shutdown();
}
}

static void race() {
    synchronized (syncObj) {
        if (!won) {
            won = true;
            System.out.println(Thread.currentThread().getName() + " won the race.");
        } else {
            System.out.println(Thread.currentThread().getName() + " lost.");
        }
    }
}
}

```

The above program synchronizes access to the `won` variable and only a single thread ever operates on the `won` variable at a time. Note that we have not marked `won` as `volatile` since the `synchronized` block establishes the *happens-before* relationship i.e. any thread reading the variables updated in the `synchronized` block see the latest value for those variables.

We can rid of the `synchronized` block by using `AtomicBoolean` as shown in the following widget:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.atomic.AtomicBoolean;

class Demonstration {

    static AtomicBoolean won = new AtomicBoolean(false);

    public static void main( String args[] ) throws Exception {
        ExecutorService es = Executors.newFixedThreadPool(25);
        try {
            int numThreads = 15;
            Runnable[] racers = new Runnable[numThreads];
            Future[] futures = new Future[numThreads];

            // create thread tasks
            for (int i = 0; i < numThreads; i++) {
                racers[i] = new Runnable() {
                    @Override
                    public void run() {
                        race();
                    }
                };
            }

            // submit threads for execution
            for (int i = 0; i < numThreads; i++) {
                futures[i] = es.submit(racers[i]);
            }

            // wait for threads to finish
            for (int i = 0; i < numThreads; i++) {
                futures[i].get();
            }
        } finally {
            es.shutdown();
        }
    }

    static void race() {
        if (won.compareAndSet(false, true)) {
            System.out.println(Thread.currentThread().getName() + " won the race.");
        } else {
            System.out.println(Thread.currentThread().getName() + " lost.");
        }
    }
}
```

```
    }  
  
}
```

The method `compareAndSet(expectedValue, updatedValue)` atomically checks if the variable `won` is false and sets it to true and then returns true if the entire operation is successful. Since these compound steps occur atomically, we can be assured that only a single thread ever gets to set the variable `won` to true. Other threads all see the variable `won` as false and fall in the `else` clause. Using `AtomicBoolean` instead of `synchronized` block results in code that is more readable, concise and faster on machines with support for machine-level atomic operations.

In summary, you can think of the `AtomicBoolean` and its sibling classes as offering `volatile`-like capabilities and then some.

Consider the following statement and pick the correct statement:

```
static volatile AtomicBoolean boolVar;
```

A)

The statement won't compile as `AtomicBoolean` already mirrors the `volatile` functionality.

B)

The `volatile` is superfluous and removing it doesn't break any functionality.

C)

`volatile` is needed for synchronization purposes.

Consider the following snippet of code:

```
AtomicBoolean flag = new AtomicBoolean(false);  
  
void method1() {  
    flag.set(false);  
}  
  
void method2() {  
    while (!flag.compareAndSet(false, true)) {  
        // ... wait  
    }  
}
```

What will be the outcome if one thread executes `method()` and another executes `method2()` ?

A)

One of the threads may get stuck in an infinite loop.

B)

The variable `flag`'s value is true at the end of the program.

C)

Only one thread is able to lock and unlock a critical section.

D)

An exception is thrown by the thread executing `method2()`

AtomicInteger

Comprehensive guide to AtomicInteger.

Overview#

The `AtomicInteger` class represents an integer value that can be updated atomically, i.e. the read-modify-write operation can be executed atomically upon an instance of `AtomicInteger`. The class extends `Number`.

`AtomicInteger` makes for great counters as it uses the compare-and-swap (CAS) instruction under the hood which doesn't penalize threads competing for access to the same data with suspension as locks do. In general, suspension and resumption of threads involves significant overhead and under low to moderate contention non-blocking algorithms that use CAS outperform lock-based alternatives.

You can find more details on non-blocking synchronization and atomics here.

Performance#

To demonstrate the performance of `AtomicInteger` we can construct a crude test, where a counter is incremented a million times by ten threads to reach a total of ten million. We'll time the run for an `AtomicInteger` counter and an ordinary `int` counter. The widget below outputs the results:

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

```

class Demonstration {

    static AtomicInteger counter = new AtomicInteger(0);
    static int simpleCounter = 0;

    public static void main( String args[] ) throws Exception {
        test(true);
        test(false);
    }

    synchronized static void incrementSimpleCounter() {
        simpleCounter++;
    }

    static void test(boolean isAtomic) throws Exception {
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        long start = System.currentTimeMillis();

        try {
            for (int i = 0; i < 10; i++) {
                executorService.submit(new Runnable() {
                    @Override
                    public void run() {
                        for (int i = 0; i < 1000000; i++) {

                            if (isAtomic) {
                                counter.incrementAndGet();
                            } else {
                                incrementSimpleCounter();
                            }
                        }
                    }
                });
            }
        } finally {
            executorService.shutdown();
            executorService.awaitTermination(1, TimeUnit.HOURS);
        }

        long timeTaken = System.currentTimeMillis() - start;
        System.out.println("Time taken by " + (isAtomic ? "atomic integer counter " : "integer
counter ") + timeTaken + " milliseconds.");
    }
}

```

Difference with **int#**

Remember that `AtomicInteger` isn't equivalent to `int`. Specifically, `AtomicInteger` class doesn't override `equals()` or `hashCode()` and each instance is distinct. `AtomicInteger` can't be used as a drop-in replacement for an `int` or `Integer`. This is demonstrated by the widget below:

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

class Demonstration {

    public static void main( String args[] ) {
        // create map
        HashMap<AtomicInteger, String> mapAtomic = new HashMap<>();
        HashMap<Integer, String> mapInt = new HashMap<>();

        // create two instances with the same value 5
        AtomicInteger fiveAtomic = new AtomicInteger(5);
        AtomicInteger fiveAtomicToo = new AtomicInteger(5);

        // create two Integer instances
        Integer fiveInt = new Integer(5);
        Integer fiveIntToo = new Integer(5);

        // Though the key is 5, but the two AtomicInteger instances
        // have different hashcodes
        mapAtomic.put(fiveAtomic, "first five atomic");
        mapAtomic.put(fiveAtomicToo, "second five atomic");
        System.out.println("value for key 5 : " + mapAtomic.get(fiveAtomic));

        // With Integer type key, the second put overwrites the
        // key with Integer value 5.
        mapInt.put(fiveInt, "first five int");
        mapInt.put(fiveIntToo, "second five int");
        System.out.println("value for key 5 : " + mapInt.get(fiveInt));
    }
}
```

Using `AtomicInteger` to simulate atomic byte#

Primitive scalars are only available as `AtomicInteger` or `AtomicLong` but we can use `AtomicInteger` to create types that simulate atomic byte. The byte data

type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive).

In the following widget we create a class `AtomicByte` which internally uses an integer to store a byte representation. The class offers two methods `shiftLeft()` and `shiftRight()` that move the bit pattern by one bit left or right respectively in a thread-safe manner and without using locks. The listing appears with comments to explain the working of the class.

Main.java

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

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

        AtomicByte atomicByte = new AtomicByte((byte) 0b11111111);
        ExecutorService executorService = Executors.newFixedThreadPool(10);

        // We'll create seven threads to shift our initial pattern of all 1s
        // to the left by one. Eventually, we should see the pattern 10000000
        // i.e. a single one followed by seven zeros. We create seven threads
        // and each thread moves the pattern to the left by one.
        try {
            for (int i = 0; i < 7; i++) {
                executorService.submit(new Runnable() {
                    @Override
                    public void run() {
                        atomicByte.shiftLeft();
                    }
                });
            }
        } finally {
            executorService.shutdown();
            executorService.awaitTermination(1, TimeUnit.HOURS);
        }

        // print after the shift operations are complete. The
        // result should be 10000000
        atomicByte.print();

    }
}
```

AtomicByte.java

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

class AtomicByte {

    // Atomic integer to store byte representation
    private AtomicInteger byteRepresentation;

    // constructor
    public AtomicByte(byte value) {
        byteRepresentation = new AtomicInteger(value);
    }

    // thread-safe method to shift the byte to the right by one bit.
    // The leftmost bit is replaced with a zero.
    public void shiftRight() {
        byte currentVal;
        do {
            currentVal = byteRepresentation.byteValue();
        } while (!byteRepresentation.compareAndSet(currentVal, currentVal >> 1));
    }

    // thread-safe method to shift the byte to the left by one bit.
    // The rightmost bit is replaced with a zero.
    public void shiftLeft() {
        byte currentVal;
        do {
            currentVal = byteRepresentation.byteValue();
        } while (!byteRepresentation.compareAndSet(currentVal, currentVal << 1));
    }

    // print the current representation of the byte.
    public void print() {
        byte currentValue = byteRepresentation.byteValue();
        byte mask = (byte) 0b10000000;

        StringBuilder sb = new StringBuilder();

        for (int i = 0; i < 8; i++) {
            sb.append((currentValue & mask) == mask ? "1" : "0");
            currentValue = (byte) (currentValue << 1);
        }

        System.out.println(sb.toString());
    }
}
```

```
}
```

The class `AtomicByte` can be retrofitted with more fancy bit manipulation functionality if desired and without involving any locks.

Using `AtomicInteger` to simulate atomic `float`#

An equivalent atomic class for `float` primitive type doesn't exist, however, we can simulate one using `AtomicInteger` as we did for the primitive type `byte`. The `float` data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion. `Float` can be used instead of type `double` if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the `java.math.BigDecimal` class instead.

To create our custom `AtomicFloat` type, we extend from the class `Number` and override several of its functions. Under the hood, we save the floating point's bit representation in an instance of `AtomicInteger`. The class listing appears below with comments for explanation:

Main.java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

class Demonstration {

    public static void main( String args[] ) throws Exception {
        AtomicFloat atomicFloat = new AtomicFloat(0.23f);
        ExecutorService executorService = Executors.newFixedThreadPool(100);

        // create 50 threads that each adds 0.17 a 1000 times. At the end our atomic
        // float should sum up to 8500.23 but you'll see a close enough value since
        // float isn't precise. However, multiple runs of the program should produce
        // the same value indicating the class AtomicFloat the thread-safe.
        try {
            for (int i = 0; i < 50; i++) {
                executorService.submit(new Runnable() {
                    @Override
```

```

        public void run() {
            for (int i = 0; i < 1000; i++)
                atomicFloat.getAndAdd(0.17f);
        }
    });
}
} finally {
    executorService.shutdown();
    executorService.awaitTermination(1, TimeUnit.HOURS);
}

// print after the shift operations are complete
System.out.println(atomicFloat.floatValue());
}
}

```

AtomicFloat.java

```

import java.util.concurrent.atomic.AtomicInteger;

class AtomicFloat extends Number {

    // integer to hold float bits
    private AtomicInteger floatRepresentation = new AtomicInteger(0);

    public AtomicFloat(float initialVal) {
        floatRepresentation.set(Float.floatToIntBits(initialVal));
    }

    @Override
    public int intValue() {
        return (int) floatValue();
    }

    @Override
    public long longValue() {
        return (long) floatValue();
    }

    @Override
    public float floatValue() {
        return Float.intBitsToFloat(floatRepresentation.get());
    }

    @Override
    public double doubleValue() {
        return (double) floatValue();
    }
}

```

```

    }

    public boolean compareAndSet(float expected, float newValue) {
        return floatRepresentation.compareAndSet(Float.floatToIntBits(expected),
Float.floatToIntBits(newValue));
    }

    public float getAndSet(float newValue) {
        return floatRepresentation.getAndSet(Float.floatToIntBits(newValue));
    }

    public float getAndAdd(float delta) {

        int currentVal;
        int newVal;
        do {
            currentVal = floatRepresentation.get();
            newVal = Float.floatToIntBits(Float.intBitsToFloat(currentVal) + delta);
        } while (!floatRepresentation.compareAndSet(currentVal, newVal));

        // Note that when we are returning the value of the float, it is possible that
        // another thread updates the float value before the following line executes. The
        // method as a whole doesn't execute atomically.
        return Float.intBitsToFloat(floatRepresentation.get());
    }
}

```

The `AtomicByte` and `AtomicFloat` classes in this lesson are shown for instructional purposes only. If you need to use these primitive types atomically, you can find well-written and well-tested libraries on the internet.

AtomicIntegerArray

Comprehensive guide to working with AtomicIntegerArray

Overview#

The class `AtomicIntegerArray` represents an array of type `int` (integers) that can be updated atomically. An instance of the `AtomicIntegerArray` can be constructed either by passing an existing array of `int` or by specifying the desired size to the constructors of `AtomicIntegerArray`. The `int` data type is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of $2^{31}-1$. In Java SE 8 and later, the `Integer` class can be used to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of $2^{32}-1$.

One notable difference between an ordinary array of `int`-s and an `AtomicIntegerArray` is that the latter provides volatile access semantics for its array elements, which isn't supported for ordinary arrays.

Example#

The code widget below demonstrates constructing an instance of `AtomicIntegerArray` and the various operations possible on it. Comments have been added to explain the various operations.

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

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

        int[] inputArray = new int[]{11, 17, 19, 23, 31};

        // use an array of ints to initialize an instance of AtomicIntegerArray
        AtomicIntegerArray atomicIntegerArray = new AtomicIntegerArray(inputArray);

        // index we'll manipulate
        int index = 3;

        // get operation
        int item = atomicIntegerArray.get(index);
        System.out.println("Item at index 3 = " + item);

        // set operation
        atomicIntegerArray.set(index, 37);
        System.out.println("Item now at index 3 = " + atomicIntegerArray.get(3));

        // compare and set operation.
        atomicIntegerArray.compareAndSet(index, 37, 41);
        System.out.println("Item at index 3 after compareAndSet operation = " +
atomicIntegerArray.get(index));

        // addAndGet() - adds the passed-in argument and returns the result
        int result = atomicIntegerArray.addAndGet(index, 5);
        System.out.println("Item at index 3 after addAndGet(5) = " + result);

        // getAndAdd() - returns the value, and then adds the passed-in argument
        result = atomicIntegerArray.getAndAdd(index, 5);
        System.out.println("Item at index 3 after addAndGet(5) = " + result);

        // getAndIncrement() - gets the value and then increments
        result = atomicIntegerArray.getAndIncrement(index);
```

```

        System.out.println("Item at index 3 after getAndIncrement() = " + result);

        // incrementAndGet() - increments and then returns the result
        result = atomicIntegerArray.incrementAndGet(index);
        System.out.println("Item at index 3 after incrementAndGet() = " + result);

        // decrementAndGet() - decrements and then gets the result
        result = atomicIntegerArray.decrementAndGet(index);
        System.out.println("Item at index 3 after decrementAndGet() = " + result);

        // getAndDecrement() -
        result = atomicIntegerArray.getAndDecrement(index);
        System.out.println("Item at index 3 after getAndDecrement() = " + result);

    }
}

```

Difference

between **AtomicIntegerArray** and and an array of **AtomicInteger**-s#

We can also create an array of **AtomicInteger**s instead of creating an **AtomicIntegerArray** but there are subtle differences between the two. These are: Creating an array of **AtomicInteger**s requires instantiating an instance of **AtomicInteger** for every index of the array, whereas in case of **AtomicIntegerArray**, we only instantiate an object of the **AtomicIntegerArray** class. In other words, using an array of **AtomicInteger**s requires an object per element whereas **AtomicIntegerArray** requires an object of the class and an array object... Both classes provide for updating the integer values present at the indexes atomically, however, in case of array of **AtomicInteger**s updating the object present at the index itself isn't thread-safe. A thread can potentially overwrite the **AtomicInteger** object at say index 0 with a new object. Such a situation isn't possible with **AtomicIntegerArray** since the class only allows **int** values to be passed-in through the public methods for updating the integer values the array holds. **AtomicInteger []** is an array of thread-safe integers, whereas **AtomicIntegerArray** is a thread-safe array of integers.

Both classes are thread-safe when multiple threads update integer values at various indexes. The following widget demonstrates ten threads randomly pick an index using **ThreadLocalRandom** and then add one to the integer value at the chosen index of an instance of **AtomicIntegerArray** and an array of **AtomicInteger**s at the same index. At the end we should observe the same

counts for all the indexes for both classes since the operations should be thread-safe.

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

class Demonstration {

    public static void main( String args[] ) throws Exception {
        final int arrayLength = 10;
        AtomicIntegerArray atomicIntegerArray = new AtomicIntegerArray(arrayLength);
        AtomicInteger[] arrayOfAtomicIntegers = new AtomicInteger[arrayLength];

        for (int i = 0; i < arrayLength; i++) {
            arrayOfAtomicIntegers[i] = new AtomicInteger(0);
        }

        ExecutorService executor = Executors.newFixedThreadPool(15);

        try {
            for (int i = 0; i < arrayLength; i++) {

                executor.submit(new Runnable() {
                    @Override
                    public void run() {

                        for (int i = 0; i < 10000; i++) {
                            // choose a random index to add to
                            int index = ThreadLocalRandom.current().nextInt(arrayLength);

                            // add one to the integer at index i
                            atomicIntegerArray.addAndGet(index, 1);
                            arrayOfAtomicIntegers[index].getAndAdd(1);
                        }
                    }
                });
            }
        } finally {
            executor.shutdown();
            executor.awaitTermination(1L, TimeUnit.HOURS);
        }

        // print the atomic integer array
        for (int i = 0; i < arrayLength; i++) {
            System.out.print(atomicIntegerArray.get(i) + " ");
        }
    }
}
```

```

        System.out.println();

        // print the array of atomic integers
        for (int i = 0; i < arrayLength; i++) {
            System.out.print(arrayOfAtomicIntegers[i].get() + " ");
        }
    }
}

```

Note that we have done the initialization of the array of `AtomicIntegers` in the main thread. The array initialization isn't thread-safe and in general the reference of the `AtomicInteger` object can be updated in a thread unsafe manner, something the `AtomicIntegerArray` doesn't suffer from.

AtomicLong

Comprehensive guide to working with AtomicLong.

Overview#

`AtomicLong` is the equivalent class for `long` type in the `java.util.concurrent.atomic` package as is `AtomicInteger` for `int` type. The `AtomicLong` class represents a long value that can be updated atomically, i.e. the read-modify-write operation can be executed atomically upon an instance of `AtomicLong`. The class extends `Number`.

Like the `AtomicInteger` class the `AtomicLong` makes for great counters, sequence numbers etc as it uses the compare-and-swap (CAS) instruction under the hood. The CAS instruction doesn't penalize competing threads for access to shared data/state with suspension as locks do. In general, suspension and resumption of threads involves significant overhead and under low to moderate contention non-blocking algorithms that use CAS outperform lock-based alternatives.

Performance#

To demonstrate the performance of `AtomicLong` we can construct a crude test, where a counter is incremented a million times by ten threads to reach a total of ten million. We'll time the run for an `AtomicLong` counter and an ordinary `long` counter. The widget below outputs the results:

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

```

```

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.*;

class Demonstration {

    static long simpleCounter;
    static AtomicLong atomicCounter;

    public static void main( String args[] ) throws Exception {
        test(true);
        test(false);
    }

    synchronized static void incrementSimpleCounter() {
        simpleCounter++;
    }

    static void test(boolean isAtomic) throws Exception {
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        long start = System.currentTimeMillis();

        try {
            for (int i = 0; i < 10; i++) {
                executorService.submit(new Runnable() {
                    @Override
                    public void run() {
                        for (int i = 0; i < 1000000; i++) {

                            if (isAtomic) {
                                atomicCounter.incrementAndGet();
                            } else {
                                incrementSimpleCounter();
                            }
                        }
                    }
                });
            }
        } finally {
            executorService.shutdown();
            executorService.awaitTermination(1, TimeUnit.HOURS);
        }

        long timeTaken = System.currentTimeMillis() - start;
        System.out.println("Time taken by " + (isAtomic ? "atomic long counter " : "long counter ") +
timeTaken + " milliseconds.");
    }
}

```

Difference with **long**#

Remember that `AtomicLong` isn't a drop-in replacement for `long`. Specifically, just like the `AtomicInteger` class, the `AtomicLong` doesn't override `equals()` or `hashcode()` and each instance is distinct. The widget below demonstrates that the same long value for two different `AtomicLong` instances doesn't hash to the same bucket.

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

class Demonstration {

    public static void main( String args[] ) {
        // create map
        HashMap<AtomicLong, String> mapAtomic = new HashMap<>();
        HashMap<Long, String> mapLong = new HashMap<>();

        // create two instances with the same long value 5
        AtomicLong fiveAtomic = new AtomicLong(5);
        AtomicLong fiveAtomicToo = new AtomicLong(5);

        // create two long instances
        Long fiveLong = new Long(5);
        Long fiveLongToo = new Long(5);

        // Though the key is 5, but the two AtomicInteger instances
        // have different hashcodes
        mapAtomic.put(fiveAtomic, "first five atomic");
        mapAtomic.put(fiveAtomicToo, "second five atomic");
        System.out.println("value for key 5 : " + mapAtomic.get(fiveAtomic));

        // With Integer type key, the second put overwrites the
        // key with Integer value 5.
        mapLong.put(fiveLong, "first five long");
        mapLong.put(fiveLongToo, "second five long");
        System.out.println("value for key 5 : " + mapLong.get(fiveLong));
    }
}
```

Using **AtomicLong** to simulate **atomic double**#

Previously, we demonstrated how the `AtomicInteger` class can be used to create atomic byte and atomic float types. Similarly, an equivalent atomic class for `double` primitive type doesn't exist, however, we can simulate one using `AtomicLong` as we did for the primitive type `byte`. The `double` data type

is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion. This data type should never be used for precise values, such as currency. For that, you will need to use the `java.math.BigDecimal` class instead.

To create our custom `AtomicDobule` type, we extend from the class `Number` and override several of its functions. Under the hood, we save the floating point's bit representation in an instance of `AtomicLong`. The class listing appears below with comments for explanation:

Main.java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicLong;

class Demonstration {

    public static void main( String args[] ) throws Exception {
        AtomicDouble atomicDouble = new AtomicDouble(0.23d);
        ExecutorService executorService = Executors.newFixedThreadPool(100);

        // create 50 threads that each adds 0.17 a 1000 times. At the end our atomic
        // double should sum up to 8500.23 but you'll see a close enough value since
        // double isn't precise. However, multiple runs of the program should produce
        // the same value indicating the class AtomicDouble the thread-safe.
        try {
            for (int i = 0; i < 50; i++) {
                executorService.submit(new Runnable() {
                    @Override
                    public void run() {
                        for (int i = 0; i < 1000; i++)
                            atomicDouble.getAndAdd(0.17d);
                    }
                });
            }
        } finally {
            executorService.shutdown();
            executorService.awaitTermination(1, TimeUnit.HOURS);
        }

        // print after the shift operations are complete
        System.out.println(atomicDouble.doubleValue());
    }
}
```

AtomicDouble.java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicLong;

class AtomicDouble extends Number {

    // long to hold double bits
    private AtomicLong doubleRepresentation = new AtomicLong(0);

    public AtomicDouble(double initialVal) {
        doubleRepresentation.set(Double.doubleToLongBits(initialVal));
    }

    @Override
    public int intValue() {
        return (int) doubleValue();
    }

    @Override
    public long longValue() {
        return (long) doubleValue();
    }

    @Override
    public float floatValue() {
        return (float) doubleValue();
    }

    @Override
    public double doubleValue() {
        return Double.longBitsToDouble(doubleRepresentation.get());
    }

    public boolean compareAndSet(double expected, double newValue) {
        return doubleRepresentation.compareAndSet(Double.doubleToLongBits(expected),
Double.doubleToLongBits(newValue));
    }

    public double getAndSet(double newValue) {
        return doubleRepresentation.getAndSet(Double.doubleToLongBits(newValue));
    }

    public double getAndAdd(double delta) {

        long currentVal;
        long newVal;
        do {
```

```

        currentVal = doubleRepresentation.get();
        newVal = Double.doubleToLongBits((Double.longBitsToDouble(currentVal) + delta));
    } while (!doubleRepresentation.compareAndSet(currentVal, newVal));

    // Note that when we are returning the value of the double, it is possible that
    // another thread updates the double value before the following line executes. The
    // method as a whole doesn't execute atomically.
    return Double.longBitsToDouble(doubleRepresentation.get());
}
}

```

The `AtomicDouble` class in this lesson is shown for instructional purposes only. If you need to use this primitive type atomically, you can find well-written and well-tested libraries on the internet. For instance, Google's Guava library offers an `AtomicDouble` type.

AtomicLongArray

Overview#

The `AtomicLongArray` is equivalent of the `AtomicIntegerArray` for the long type. You'll observe several similarities between the two classes.

The class `AtomicLongArray` represents an array of type `long` that can be updated atomically. We can use long when the range of values we want to represent falls outside the range of values for an integer. An instance of the `AtomicLongArray` can be constructed either by passing an existing array of `long` or by specifying the desired size to the constructors of `AtomicLongArray`. The `long` data type is a 64-bit signed two's complement integer, which has a minimum value of -2^{63} and a maximum value of $2^{63}-1$. In Java SE 8 and later, `Long` class can be used to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of $2^{64}-1$.

One notable difference between an ordinary array of `long`-s and an `AtomicLongArray` is that the latter provides volatile access semantics for its array elements, which isn't supported for ordinary arrays.

Example#

The code widget below demonstrates constructing an instance of `AtomicLongArray` and the various operations possible on it. Comments have been added to explain the various operation

```
import java.util.concurrent.atomic.AtomicLong;
```

```
import java.util.concurrent.atomic.AtomicLongArray;

class Demonstration {

    public static void main( String args[] ) {
        long[] inputArray = new long[]{11, 17, 19, 23, 31};

        // use an array of ints to initialize an instance of AtomicIntegerArray
        AtomicLongArray atomicIntegerArray = new AtomicLongArray(inputArray);

        // index we'll manipulate
        int index = 3;

        // get operation
        long item = atomicIntegerArray.get(index);
        System.out.println("Item at index 3 = " + item);

        // set operation
        atomicIntegerArray.set(index, 37);
        System.out.println("Item now at index 3 = " + atomicIntegerArray.get(3));

        // compare and set operation.
        atomicIntegerArray.compareAndSet(index, 37, 41);
        System.out.println("Item at index 3 after compareAndSet operation = " +
atomicIntegerArray.get(index));

        // addAndGet() - adds the passed-in argument and returns the result
        long result = atomicIntegerArray.addAndGet(index, 5);
        System.out.println("Item at index 3 after addAndGet(5) = " + result);

        // getAndAdd() - returns the value, and then adds the passed-in argument
        result = atomicIntegerArray.getAndAdd(index, 5);
        System.out.println("Item at index 3 after addAndGet(5) = " + result);

        // getAndIncrement() - gets the value and then increments
        result = atomicIntegerArray.getAndIncrement(index);
        System.out.println("Item at index 3 after getAndIncrement() = " + result);

        // incrementAndGet() - increments and then returns the result
        result = atomicIntegerArray.incrementAndGet(index);
        System.out.println("Item at index 3 after incrementAndGet() = " + result);

        // decrementAndGet() - decrements and then gets the result
        result = atomicIntegerArray.decrementAndGet(index);
        System.out.println("Item at index 3 after decrementAndGet() = " + result);

        // getAndDecrement() -
        result = atomicIntegerArray.getAndDecrement(index);
```

```

        System.out.println("Item at index 3 after getAndDecrement() = " + result);
    }
}

```

Difference between **AtomicLongArray** and and an array of **AtomicLong**-s#

We can also create an array of **AtomicLong**-s instead of creating an **AtomicLongArray** but there are subtle differences between the two. These are: Creating an array of **AtomicLong**-s requires instantiating an instance of **AtomicLong** for every index of the array, whereas in case of **AtomicLongArray**, we only instantiate an object of the **AtomicLongArray** class. In other words, using an array of **AtomicLong**-s requires an object per element whereas **AtomicLongArray** requires an object of the class and an array object... Both classes provide for updating the long values present at the indexes atomically, however, in case of array of **AtomicLong**-s updating the object present at the index itself isn't thread-safe. A thread can potentially overwrite the **AtomicLong** object at say index 0 with a new object. Such a situation isn't possible with **AtomicLongArray** since the class only allows **long** values to be passed-in through the public methods for updating the long values the array holds. **AtomicLong []** is an array of thread-safe longs, whereas **AtomicLongArray** is a thread-safe array of longs.

Both classes are thread-safe when multiple threads update long values at various indexes. The following widget demonstrates ten threads randomly pick an index using **ThreadLocalRandom** and then add one to the long value at the chosen index of an instance of **AtomicLongArray** and an array of **AtomicLong**-s at the same index. At the end we should observe the same counts for all the indexes for both classes since the operations should be thread-safe.

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicLong;
import java.util.concurrent.atomic.AtomicLongArray;

class Demonstration {

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

        final int arrayLength = 10;
        AtomicLongArray atomicLongArray = new AtomicLongArray(arrayLength);
    }
}

```

```
AtomicLong[] arrayOfAtomicLongs = new AtomicLong[arrayLength];

for (int i = 0; i < arrayLength; i++) {
    arrayOfAtomicLongs[i] = new AtomicLong(0);
}

ExecutorService executor = Executors.newFixedThreadPool(15);

try {

    for (int i = 0; i < arrayLength; i++) {

        executor.submit(new Runnable() {
            @Override
            public void run() {

                for (int i = 0; i < 10000; i++) {
                    // choose a random index to add to
                    int index = ThreadLocalRandom.current().nextInt(arrayLength);

                    // add one to the integer at index i
                    atomicLongArray.addAndGet(index, 1);
                    arrayOfAtomicLongs[index].getAndAdd(1);
                }
            }
        });
    }

} finally {
    executor.shutdown();
    executor.awaitTermination(1L, TimeUnit.HOURS);
}

// print the atomic integer array
for (int i = 0; i < arrayLength; i++) {
    System.out.print(atomicLongArray.get(i) + " ");
}

System.out.println();

// print the array of atomic integers
for (int i = 0; i < arrayLength; i++) {
    System.out.print(arrayOfAtomicLongs[i].get() + " ");
}
}
```

Note that we have done the initialization of the array of `AtomicLong`-s in the main thread. The array initialization isn't thread-safe and in general the reference of the `AtomicLong` object can be updated in a thread unsafe manner, something the `AtomicLongArray` doesn't suffer from.

LongAdder

Learn why LongAdder can be a better choice for computing statistics in a multithreaded environment than `AtomicLong`.

Overview#

Consider a scenario where you are hosting an API service and need to maintain a running count of all the invocations of a particular API endpoint. You could use `AtomicLong` for the occasion as a counter but if you expect several concurrent requests hitting your endpoint, then multiple threads will attempt to increment the `AtomicLong` instance. In such a scenario where the intention is to compute statistics a better approach is to use the `LongAdder` class rather than the `AtomicLong`.

`AtomicLong` internally uses the compare and set instruction to perform the increment operation. In case of high contention, that is multiple threads attempting to increment an instance, a single thread wins the race to perform the increment operation while the rest have to retry until all of them succeed one by one. `AtomicLong` doesn't use locks and threads don't suspend accessing it. We discuss non-blocking synchronization to implement thread-safe algorithms and cons of locking in the Atomic Classes section.

`LongAdder` overcomes high contention by keeping an array of counts, which can be thought of as variants of `AtomicLong`, and each one of them can be incremented atomically and independently of the other. The contention is spread out across the array resulting in increased throughput. Though the increased throughput comes at the cost of using more space. When the final value is requested, the sum of the array is calculated and returned by invoking the `sum()` method. Note that `sum()` isn't an atomic snapshot and any writes by threads while `sum()` is executing may not be reflected in the result. An accurate result is returned when concurrent updates don't occur.

The counts are of a non-public type `cell` which is a variant of `AtomicLong`. The array is referred to as the *table of Cells* and it grows in size as contention increases but its maximum size is capped by the number of CPUs.

Example#

The widget below runs a crude performance test between `LongAdder` and `AtomicLong`. We create ten threads that increment an instance of either class a million times. The sum at the end should come to be ten million. From the test set-up we can see that the instance of each class is highly contended for. From the test output we can see that `LongAdder` performs much better than `AtomicLong`.

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

class Demonstration {

    static AtomicLong atomicLong = new AtomicLong(0);
    static LongAdder longAdder = new LongAdder();

    public static void main( String args[] ) throws Exception {
        withAtomicLong();
        withLongAdder();
    }

    static void withAtomicLong() throws Exception {

        ExecutorService executorService = Executors.newFixedThreadPool(15);
        long start = System.currentTimeMillis();

        try {
            for (int i = 0; i < 10; i++) {
                executorService.submit(new Runnable() {
                    @Override
                    public void run() {
                        for (int i = 0; i < 1000000; i++) {
                            atomicLong.incrementAndGet();
                        }
                    }
                });
            }
        } finally {
            executorService.shutdown();
            executorService.awaitTermination(1, TimeUnit.HOURS);
        }

        long timeTaken = System.currentTimeMillis() - start;
        System.out.println("Time taken by atomic long " + timeTaken + " milliseconds and count " +
atomicLong.get());
    }
}
```

```

static void withLongAddr() throws Exception {

    ExecutorService executorService = Executors.newFixedThreadPool(15);
    long start = System.currentTimeMillis();

    try {
        for (int i = 0; i < 10; i++) {
            executorService.submit(new Runnable() {
                @Override
                public void run() {
                    for (int i = 0; i < 1000000; i++) {
                        longAdder.increment();
                    }
                }
            });
        }
    } finally {
        executorService.shutdown();
        executorService.awaitTermination(1, TimeUnit.HOURS);
    }
    long timeTaken = System.currentTimeMillis() - start;
    System.out.println("Time taken by long adder " + timeTaken + " milliseconds and count = "
+ longAdder.sum());
}
}

```

LongAccumulator

Comprehensive guide to working with LongAccumulator.

Overview#

The `LongAccumulator` class is similar to the `LongAdder` class, except that the `LongAccumulator` class allows for a function to be supplied that contains the logic for computing results for accumulation. In contrast to `LongAdder`, we can perform a variety of mathematical operations rather than just addition. The supplied function to a `LongAccumulator` is of type `LongBinaryOperator`. The class `LongAccumulator` extends from the class `Number` but doesn't define the methods `compareTo()`, `equals()`, or `hashCode()` and shouldn't be used as keys in collections such as maps.

An example of creating an accumulator that simply adds long values presented to it

```

// function that will be supplied to an instance of LongAccumulator
LongBinaryOperator longBinaryOperator = new LongBinaryOperator() {
    @Override
    public long applyAsLong(long left, long right) {
        return left + right;
    }
};

// instantiating an instance of LongAccumulator with an initial value of zero
LongAccumulator longAccumulator = new LongAccumulator(longBinaryOperator,
, 0);

```

Note that in the above example, we have supplied a function that simply adds the new long value presented to it. The method `applyAsLong` has two operands `left` and `right`. The `left` operand is the current value of the `LongAccumulator`. In the above example, it'll be zero initially, because that is what we are passing-in to the constructor of the `LongAccumulator` instance. The code widget below runs this example and prints the operands and the final sum.

```

import java.util.concurrent.atomic.LongAccumulator;
import java.util.function.LongBinaryOperator;

class Demonstration {
    public static void main( String args[] ) {
        // function that will be supplied to an instance of LongAccumulator
        LongBinaryOperator longBinaryOperator = new LongBinaryOperator() {
            @Override
            public long applyAsLong(long left, long right) {
                System.out.println(left + " " + right);
                return left + right;
            }
        };

        // instantiating an instance of LongAccumulator with an initial value of zero
        LongAccumulator longAccumulator = new LongAccumulator(longBinaryOperator, 0);

        for (int i = 0; i < 10; i++) {
            longAccumulator.accumulate(1);
        }

        System.out.println("Final value = " + longAccumulator.get());
    }
}

```

As you can see we aren't confined to adding long values, rather we can perform as complex operations as desired in the supplied function, which

makes the `LongAccumulator` class far more versatile than the `LongAdder` class which is limited to addition. In fact, `LongAdder` can be thought of as a specialized case of `LongAccumulator` for keeping counts and sums.

Distributing contention#

We can achieve the same functionality by using an instance of `AtomicLong` as we can with the `LongAccumulator`, however, the rational for `LongAccumulator` is to distribute contention among threads by maintaining a set of variables that grow dynamically and each one is updated by only a subset of threads. Thus the contention is spread from a single variable to several variables. When the current value is asked for by invoking the `get()` or the `longValue()` methods, all the underlying variables are accumulated by applying the supplied function and the result is returned. The expected throughput of `LongAccumulator` is significantly higher when used in place of `AtomicLong` under high contention. The improved performance comes at the cost of using more space.

Order of accumulation#

When multiple threads accumulate an instance of `LongAccumulator`, eventually all the long values in the underlying set are accumulated using the supplied function. The order in which these long values are accumulated isn't guaranteed and the supplied function should produce the same value irrespective of the order in which these values are accumulated. In case, the supplied function isn't commutative i.e., `left + right` isn't the same as `right + left` then the accumulation can produce different results for the same series of accumulated long values.

Example#

In the example below, we use the `LongAccumulator` class to keep track of the maximum value observed. There are several threads that use the `ThreadLocalRandom` class to produce a random long value less than 1000, and then attempt to update the instance of `LongAccumulator`. We conduct the same test using `AtomicLong` and time the two tests. Go through the listing which is self-explanatory.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicLong;
import java.util.concurrent.atomic.LongAccumulator;
import java.util.function.LongBinaryOperator;
```

```

class Demonstration {

    static int numThreads = 2;
    static int poolSize = 50;
    static int iterations = 10000;

    public static void main( String args[] ) throws Exception {
        testWithLongAccumulator();
        testWithAtomicLong();
    }

    static void testWithLongAccumulator() throws Exception {

        // function that will be supplied to an instance of LongAccumulator
        LongBinaryOperator longBinaryOperator = new LongBinaryOperator() {
            @Override
            public long applyAsLong(long left, long right) {
                return left > right ? left : right;
            }
        };

        // instantiating an instance of LongAccumulator with the lowest possible min value
        LongAccumulator longAccumulator = new LongAccumulator(longBinaryOperator,
        Long.MIN_VALUE);

        ExecutorService executorService = Executors.newFixedThreadPool(poolSize);
        long start = System.currentTimeMillis();

        try {
            for (int i = 0; i < numThreads; i++) {
                executorService.submit(new Runnable() {
                    @Override
                    public void run() {
                        for (int j = 0; j < iterations; j++) {
                            long value = ThreadLocalRandom.current().nextLong(1000);
                            longAccumulator.accumulate(value);
                        }
                    }
                });
            }
        } finally {
            executorService.shutdown();
            executorService.awaitTermination(1, TimeUnit.HOURS);
        }

        long timeTaken = System.currentTimeMillis() - start;
        System.out.println("Time taken by LongAccumulator " + timeTaken + " milliseconds and
max value observed = " + longAccumulator.get());
    }
}

```

```

}

static void testWithAtomicLong() throws Exception {

    int numThreads = 20;

    AtomicLong atomicLong = new AtomicLong(Long.MIN_VALUE);

    ExecutorService executorService = Executors.newFixedThreadPool(poolSize);
    long start = System.currentTimeMillis();

    try {
        for (int i = 0; i < numThreads; i++) {
            executorService.submit(new Runnable() {
                @Override
                public void run() {
                    for (int j = 0; j < iterations; j++) {
                        long value = ThreadLocalRandom.current().nextLong(1000);

                        long currentMax;

                        do {
                            currentMax = atomicLong.get();
                            if (currentMax > value)
                                break;
                        } while (!atomicLong.compareAndSet(currentMax, value));
                    }
                }
            });
        }
    } finally {
        executorService.shutdown();
        executorService.awaitTermination(1, TimeUnit.HOURS);
    }

    long timeTaken = System.currentTimeMillis() - start;
    System.out.println("Time taken by AtomicLong " + timeTaken + " milliseconds and max
value observed = " + atomicLong.get());
}
}

```

Note that the above test is crude and imprecise in nature, but does give a general idea of the performance of the two classes under high contention. We can tweak the different parameters such as the number of `iterations` or `numThreads` to produce an environment with different contention characteristics and maybe different results.

DoubleAdder

Guide to using DoubleAdder class.

Overview#

The `DoubleAdder` class offers an alternative mechanism to using atomic classes or lock-based counters in situations where a variable is repeatedly updated to compute stats or summary counts. In such scenarios throughput suffers significantly when several threads attempt to update a single variable. In case of locks, threads get suspended and resumed while in case of atomic classes threads spin until successful. With `DoubleAdder` the idea is to have several variables that maintain the count in a thread-safe manner rather than a single variable, thus effectively reducing contention. Reducing contention improves throughput though at the cost of using more space.

The `DoubleAdder` class is very similar to `LongAdder`, in that both classes maintain a set of variables, of a package-private type `java.util.concurrent.atomic.Striped64.Cell` which is a variant of `AtomicLong`. As the number of threads manipulating an instance of `DoubleAdder` increases, this underlying table of `Cells` keeps growing upto the maximum number of CPUs. The long to double conversions are handled internally. The array of counts helps to spread contention among threads rather than all threads competing to increment/decrement a single instance.

When the cumulative or final count is desired, the `sum()` method can be invoked. Floating point arithmetic isn't associative and when `sum()` is invoked it doesn't guarantee the order in which the accumulation happens. This appears as a disclaimer in the class's documentation stating *The order of accumulation within or across threads is not guaranteed. Thus, this class may not be applicable if numerical stability is required, especially when combining values of substantially different orders of magnitude..* Additionally, `sum()` is not an atomic snapshot, and if concurrent updates by threads continue while `sum()` is executing, the newer updates may not be reflected in the result. However, when `sum()` is invoked in the absence of concurrent updates i.e. no thread is performing a write operation, then `sum()` returns an accurate result.

`DoubleAdder` should be used in scenarios where updates to an instance of `DoubleAdder` are frequent and reads are rare. For instance, calculating summary statistics where several threads update a common variable is a good candidate for use of `DoubleAdder`.

Example#

In the example below, we have ten threads that attempt to increment an instance of `DoubleAdder` by one a million times each. Though we don't have an equivalent for double in the atomic classes but for purposes of our crude test, we use an `AtomicLong` that is also incremented by one and then time the two runs, one with `AtomicLong` and one with `DoubleAdder`. The test is setup so that the counter variables in both scenarios experience high contention and the results show that `DoubleAdder` outperforms `AtomicLo`

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

class Demonstration {

    static DoubleAdder doubleAdder = new DoubleAdder();
    static AtomicLong atomicLong = new AtomicLong(0);

    public static void main(String[] args) throws Exception {
        withAtomicLong();
        withDoubleAdder();
    }

    static void withAtomicLong() throws Exception {

        ExecutorService executorService = Executors.newFixedThreadPool(15);
        long start = System.currentTimeMillis();

        try {
            for (int i = 0; i < 10; i++) {
                executorService.submit(new Runnable() {
                    @Override
                    public void run() {
                        for (int i = 0; i < 1000000; i++) {
                            atomicLong.incrementAndGet();
                        }
                    }
                });
            }
        } finally {
            executorService.shutdown();
            executorService.awaitTermination(1, TimeUnit.HOURS);
        }

        long timeTaken = System.currentTimeMillis() - start;
        System.out.println("Time taken by atomic long " + timeTaken + " milliseconds and count " +
atomicLong.get());
    }
}
```

```

}

static void withDoubleAddr() throws Exception {

    ExecutorService executorService = Executors.newFixedThreadPool(15);
    long start = System.currentTimeMillis();

    try {
        for (int i = 0; i < 10; i++) {
            executorService.submit(new Runnable() {
                @Override
                public void run() {
                    for (int i = 0; i < 1000000; i++) {
                        doubleAdder.add(1.00d);
                    }
                }
            });
        }
    } finally {
        executorService.shutdown();
        executorService.awaitTermination(1, TimeUnit.HOURS);
    }
    long timeTaken = System.currentTimeMillis() - start;
    System.out.println("Time taken by double adder " + timeTaken + " milliseconds and count =
" + doubleAdder.sum());
}
}

```

DoubleAccumulator

Comprehensive guide to working with DoubleAccumulator class.

Overview#

The `DoubleAccumulator` class is similar to the `DoubleAdder` class, except that the `DoubleAccumulator` class allows for a function to be supplied that contains the logic for computing results for accumulation. In contrast to `DoubleAdder`, we can perform a variety of mathematical operations rather than just addition. The supplied function to a `DoubleAccumulator` is of type `DoubleBinaryOperator`. The class `DoubleAccumulator` extends from the class `Number` but doesn't define the methods `compareTo()`, `equals()`, or `hashCode()` and instances of the class shouldn't be used as keys in collections such as maps.

An example of creating an accumulator that simply adds double values presented to it appears below:

```
// function that will be supplied to an instance of DoubleAccumulator
DoubleBinaryOperator doubleBinaryOperator = new DoubleBinaryOperator() {
    @Override
    public double applyAsDouble(double left, double right) {
        return left + right;
    }
};

// instantiating an instance of DoubleAccumulator with an initial value of zero
DoubleAccumulator longAccumulator = new DoubleAccumulator(doubleBinaryOperator, 0);
```

Note that in the above example, we have supplied a function that simply adds the new double value presented to it. The method `applyAsDouble` has two operands `left` and `right`. The `left` operand is the current value of the `DoubleAccumulator`. In the above example, it'll be zero initially, because that is what we are passing-in to the constructor of the `DoubleAccumulator` instance. The code widget below runs this example and prints the operands and the final sum.

```
import java.util.concurrent.atomic.DoubleAccumulator;
import java.util.function.DoubleBinaryOperator;

class Demonstration {

    public static void main( String args[] ) {

        // function that will be supplied to an instance of DoubleAccumulator
        DoubleBinaryOperator doubleBinaryOperator = new DoubleBinaryOperator() {
            @Override
            public double applyAsDouble(double left, double right) {
                System.out.println(left + " " + right);
                return left + right;
            }
        };

        // instantiating an instance of DoubleAccumulator with an initial value of zero
        DoubleAccumulator doubleAccumulator = new
        DoubleAccumulator(doubleBinaryOperator, 0);

        for (int i = 0; i < 10; i++) {
            doubleAccumulator.accumulate(1);
        }
    }
}
```

```
        System.out.println("Final value = " + doubleAccumulator.get());
    }
}
```

As you can see we aren't confined to adding double values, rather we can perform as complex operations as desired in the supplied function, which makes the `DoubleAccumulator` class far more versatile than the `DoubleAdder` class which is limited to addition. In fact, `DoubleAdder` can be thought of as a specialized case of `DoubleAccumulator` which adds double values.

Distributing contention#

We can achieve the same functionality by using an instance of `AtomicLong` as we can with the `DoubleAccumulator`, however, the rationale for `DoubleAccumulator` is to distribute contention among threads by maintaining a set of variables that grow dynamically and each one is updated by only a subset of threads. Thus the contention is spread from a single variable to several variables. When the current value is asked for by invoking the `get()` or the `doubleValue()` methods, all the underlying variables are accumulated by applying the supplied function and the result is returned. The expected throughput of `DoubleAccumulator` is significantly higher when used in place of `AtomicLong` for maintaining a double value under high contention. The improved performance comes at the cost of using more space. Additionally, we'll also have the overhead of converting double bytes to long bytes and back when using `AtomicLong` for maintaining the running double value. See the `AtomicLong` lesson where we discuss using `AtomicLong` for double values.

Order of accumulation#

When multiple threads accumulate an instance of `DoubleAccumulator`, eventually all the double values in the underlying set are accumulated using the supplied function. The order in which these double values are accumulated isn't guaranteed and the supplied function should produce the same value irrespective of the order in which these values are accumulated. In case, the supplied function isn't commutative i.e., `left + right` isn't the same as `right + left` then the accumulation can produce different results for the same series of accumulated double values. The accumulation taking place in an arbitrary order may make this class unsuitable for scenarios where numerical stability is required, especially when combining values of very different orders of magnitude.

Example#

In the example below, we use the `DoubleAccumulator` class to keep track of the maximum value observed. There are several threads that use the `ThreadLocalRandom` class to produce a random double value less than 1000, and then attempt to update the instance of `DoubleAccumulator`. Go through the listing which is self-explanatory.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.DoubleAccumulator;
import java.util.function.DoubleBinaryOperator;

class Demonstration {

    static int numThreads = 2;
    static int poolSize = 50;
    static int iterations = 10000;

    public static void main( String args[] ) throws Exception {
        // function that will be supplied to an instance of LongAccumulator
        DoubleBinaryOperator doubleBinaryOperator = new DoubleBinaryOperator() {
            @Override
            public double applyAsDouble(double left, double right) {
                return left > right ? left : right;
            }
        };

        // instantiating an instance of LongAccumulator with the lowest possible min value
        DoubleAccumulator doubleAccumulator = new
        DoubleAccumulator(doubleBinaryOperator, Double.MIN_VALUE);
        ExecutorService executorService = Executors.newFixedThreadPool(poolSize);

        try {
            for (int i = 0; i < numThreads; i++) {
                executorService.submit(new Runnable() {
                    @Override
                    public void run() {
                        for (int j = 0; j < iterations; j++) {
                            double value = ThreadLocalRandom.current().nextDouble(1000.00);
                            doubleAccumulator.accumulate(value);
                        }
                    }
                });
            }
        } finally {
            executorService.shutdown();
            executorService.awaitTermination(1, TimeUnit.HOURS);
        }
    }
}
```

```
    }

    System.out.println("Max value observed = " + doubleAccumulator.get());
}
}
```

AtomicReference

Complete guide to understanding and effectively working with `AtomicReference` class. Learn the differences among atomic assignment of references in Java, volatile variables and the `AtomicReference` class.

Overview#

A reference type is a data type that represents an instance of a Java class, i.e. it is a type other than one of the Java's primitive types. For instance:

```
Long myLong = new Long(5);
```

In the above snippet the variable `myLong` represents a reference type. When we create the above object `myLong`, Java allocates appropriate memory for the object to be stored. The variable `myLong` points to this address in the memory, which stores the object. The address of the object in the memory is called the *reference*. Consider the below snippet:

```
Long myLong = new Long(5);
Long otherLong = myLong;
```

The two variables `myLong` and `otherLong` both point to the same memory location where the `Long` object is stored or we can say both variables are assigned the same reference (address) to the object but not the object itself.

Given the above context, it becomes easier to understand the `AtomicReference` class. According to the official documentation, `AtomicReference` allows us to atomically update the reference to an object. Before we further delve into the topic, we'll first clarify the distinction between `AtomicReference` and assignment of references atomically in Java.

Atomic assignment of references in Java#

Consider the following snippet:

```

class SomethingUseful {
    Thread thread;
    int myInt;
    HashMap<String, String> map;
    double myDouble;
    long myLong;

    void usefulFunction(int passedInt, long passedLong, double passedDouble,
        HashMap<String, String> passedMap) {
        thread = Thread.currentThread();
        myInt = passedInt;
        map = passedMap;
    }

}

```

If the method `usefulFunciton()` is invoked by several threads, can we see garbage values for the variables of an instance of class `SomethingUseful`? In the method `usefulFunciton()` there are two primitive type assignments and two reference assignments. The following holds true for assignments in Java as per the language specification:

- All reference assignments are atomic. This doesn't mean our method `usefulFunction()` is thread-safe. It just means that when a thread assigns the variables `map` or the `thread`, the assignment is atomic, i.e. it can't happen that the variable `thread` or `map` hold some bytes from the assignment operation of one thread and other bytes from the assignment operation of another thread. Whatever reference the two variables hold will reflect an assignment from one of the threads. Reference reads and writes are always atomic whether the reference itself consists of 32 or 64 bits.
- Assignments and reads for primitive data types except for `double` and `long` are always atomic. If two threads are invoking the method `usefulFunction()` and passing in 5 and 7 for the integer variable then the variable `myInt` will hold either 5 or 7 and not any other value.
- The reads and writes to `double` and `long` primitive types aren't atomic. The JVM specification allows implementations to break-up the write of the 64 bit `double` or `long` primitive type into two writes, one for each 32 bit half. This can result in a situation where two threads are simultaneously writing a `double` or `long` value and a third thread observes the first 32 bits from the write by the first thread and the next 32 bits from the write by the second thread. As a result the third thread reads a value that has neither been written by either of the two threads or is a garbage value. In order to make reads and writes to `double` or `long` primitive types atomic, we must mark

them as `volatile`. The specification guarantees writes and reads to volatile `double` and `long` primitive types as atomic. Note that some JVM implementations may make the writes and reads to `double` and `long` types as atomic but this isn't universally guaranteed across all the JVM implementations.

So far we understood what a reference in Java means and that except for two primitive types all assignments are atomic in Java. This may confuse some readers as to the purpose and intent of the class `AtomicReference` when assignments in Java are mostly atomic and the ones to `double` and `long` can be made atomic using `volatile`. We'll address this question next.

AtomicReference class#

The class `AtomicReference` promises assignment of a reference atomically, however, it offers far more in functionality than just an atomic assignment. Consider the snippet below:

```
// Regular variables
Thread thread;
int myInt;

// Assignments are indeed atomic, however, the method isn't thread-safe and the threads
// can potentially cache a stale value of the two variables in their cache.
void myFunction(int passedInt) {
    myInt = passedInt;
    thread = Thread.currentThread();
}
```

The variables `thread` and `myInt` can both be cached and a thread may not observe the latest value for them.

Next, we can mark the variable `thread` as `volatile`.

```
// Marking Thread reference as volatile
volatile Thread thread;

// No change here.
int myInt;

// Assignments are atomic, however, the method isn't thread-safe. The memory
// visibility guarantees are now different. If a thread executes the function and then
// another thread reads the Thread reference, the reading thread will also see the latest
// value for myInt even though it is not marked as volatile.
void myFunction(int passedInt) {
```

```

        myInt = passedInt;
        thread = Thread.currentThread();
    }
}

```

In the above snippet, the threads reading the `thread` variable also see the latest value for the variable `myInt` because the variable `myInt` is assigned to before the `thread` variable. Since `thread` is marked volatile it establishes the *happens-before* relations for the variables that are assigned earlier than it.

We can rewrite the above snippet using the `AtomicReference` class as follows:

```

// Using an atomic reference
AtomicReference<Thread> threadAtomicReference = new AtomicReference<>(
);
// No change here.
int myInt;

// Assignments are atomic, however, the method isn't thread-safe. The memory
// visibility guarantees are similar to the snippet with the volatile thread
// variable.
void myFunction(int passedInt) {
    myInt = passedInt;
    threadAtomicReference.set(Thread.currentThread());
}

```

The above snippet is equivalent of the snippet with `thread` marked as volatile.

Using `AtomicReference` we establish the *happen-before* relationship similar to when a `volatile` variable is written or read. Reading or writing to a volatile variable guarantees that the variable value is written to and read from the main memory and not the cache. Therefore all threads observing a volatile variable read the most up to date value for that variable and avoid using a stale value for the variable from their cache.
The `get()` and `set()` operations on the `AtomicReference` variable promise similar guarantees as the read and write of a volatile variable respectively.

However, also note that the assignment or initialization of an `AtomicReference` doesn't guarantee a happens-before relationship. For instance,

```

class SomeClass {
    AtomicReference<Object> atomicReference;

    public void init(Object obj) {
        atomicReference = new AtomicReference<Object>(obj);
    }
}

```

```
    }
    // ... Rest of class definition
}
```

In the above snippet, the `atomicReference` is being initialized or assigned to and unless the `atomicReference` variable is marked volatile, a happens-before relationship isn't established.

check-then-act#

The astute reader would question the usability of the `AtomicReference` class if we can use `volatile`, as we have done in our previous snippets, to achieve the same functionality. The crucial functionality `AtomicReference` offers is to execute algorithms or actions that check the value of a variable and then act upon it based on the observed value, also known as *check-then-act*, atomically. We slightly modify our previous snippet to create a check-then-act scenario below:

```
// Marking Thread reference as volatile
volatile Thread thread;
int myInt;

// Assignments are atomic, however, the method isn't thread-safe. No
// more
// than two threads can enter the if-clause even though the thread var-
iable
// is marked volatile. Remember volatile doesn't mean thread-safety!
void myFunction(int passedInt) {
    myInt = passedInt;
    if (thread == null) {
        thread = Thread.currentThread();
    }
}
```

In the above scenario, we intend for a single thread to find the `thread` variable null and initialize it. We want the two steps - check if `thread` is null and act assign`thread` - to be performed atomically, which `volatile` can't help us with. In fact, `volatile` is only limited to offering stronger memory visibility guarantees and that's about it. This is where `AtomicReference` comes in and allows us to execute the two steps atomically. The same snippet re-written with `AtomicReference` appears below:

```
// Using an atomic reference
AtomicReference<Thread> threadAtomicReference = new AtomicReference<>(
);
int myInt;

// Assignments are atomic, however, the method isn't thread-safe. Onl
```

```

y a
    // single thread ever executes the if-clause.
    void myFunction(int passedInt) {
        myInt = passedInt;
        if (threadAtomicReference.compareAndSet(null, Thread.currentThread
())) {
            thread = Thread.currentThread();
        }
    }
}

```

Take a moment to ponder on the above snippet and observe the following facts:

1. The method as a whole is still not thread-safe. The value of `myInt` may be from a thread that didn't initialize the `threadAtomicReference` variable.
2. If the variable `myInt` is moved **after** the `if-clause` the *happens-before* isn't established between variables `threadAtomicReference` and `myInt` and the value of `myInt` may only be updated in the cache and not the main memory. A thread that subsequently reads the variable `threadAtomicReference` will fetch all the variables visible to it from the main memory as per the volatile variable visibility guarantee but since `myInt` was never updated in the main memory, the reader thread may observe a stale value for the `myInt` variable.
3. The initialization is guaranteed to be atomic and only a single thread ever enters the `if-clause`.

Example#

In our code widgets below, we'll construct three versions of the same example. We have several threads all attempting to simultaneously initialize a variable `firstThread` to the current thread reference using `Thread.currentThread()`. If the threads find the `firstThread` variable `null` it simply initializes the variable to itself. Here's the first version using volatile variable:

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicReference;

class Demonstration {
    static volatile Thread firstThread = null;

```

```

public static void main( String args[] ) throws Exception {
    ExecutorService executor = Executors.newFixedThreadPool(50);

    try {
        for (int i = 0; i < 25; i++) {
            executor.submit(new Runnable() {
                @Override
                public void run() {

                    if (firstThread == null) {
                        firstThread = Thread.currentThread();
                        System.out.println(Thread.currentThread().getName() + " takes first turn");
                    }
                }
            });
        }
    } finally {
        executor.shutdown();
        executor.awaitTermination(1, TimeUnit.HOURS);
    }
}
}

```

If you run the above widget enough times, you'll see some runs show multiple threads entering the **if-clause**. We can make the initialization thread-safe by using a syncrhonized as shown below:

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicReference;

class Demonstration {

    static Thread firstThread = null;

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

        ExecutorService executor = Executors.newFixedThreadPool(50);
        try {

            for (int i = 0; i < 25; i++) {
                executor.submit(new Runnable() {
                    @Override
                    public void run() {
                        synchronized (this) {
                            if (firstThread == null) {
                                firstThread = Thread.currentThread();
                                System.out.println(Thread.currentThread().getName() + " takes first turn");
                            }
                        }
                    }
                });
            }
        } finally {
            executor.shutdown();
            executor.awaitTermination(1, TimeUnit.HOURS);
        }
    }
}

```

```
        }
    }
}
});
}
}
} finally {
    executor.shutdown();
    executor.awaitTermination(1, TimeUnit.HOURS);
}
}
}
```

In the above code-snippet the initialization is thread-safe and only one thread ever executes the `if-clause`. Finally, we can rewrite the above code using the `AtomicReference` class as shown below:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicReference;

class Demonstration {

    static AtomicReference<Thread> atomicReference = new AtomicReference<>();

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

        ExecutorService executor = Executors.newFixedThreadPool(50);
        try {
            for (int i = 0; i < 25; i++) {
                executor.submit(new Runnable() {
                    @Override
                    public void run() {

                        if (atomicReference.get() == null) {
                            if (atomicReference.compareAndSet(null, Thread.currentThread())) {
                                System.out.println(Thread.currentThread().getName() + " takes first turn");
                            }
                        }
                    }
                });
            }
        } finally {
            executor.shutdown();
            executor.awaitTermination(1, TimeUnit.HOURS);
        }
    }
}
```

The above code uses `AtomicReference` to initialize the static variable `atomicReference` in a thread-safe manner. Only a single thread is able to initialize the variable while all others find the variable not null. In terms of performance, under low to moderate contention `AtomicReference` is expected to perform better than the same code written using `synchronized` or other locking mechanisms.

Conclusion#

Generally speaking, it is easy to get confused among the three concepts atomic reference/primitive assignments, `volatile` and `AtomicReference`. It is important to understand the distinction between assignment of reference and primitive types being atomic and the class `AtomicReference`. Additionally, `AtomicReference` goes beyond `volatile` and allows for executing check-then-act type scenarios to be executed atomically.

AtomicReferenceArray

Guide to working with `AtomicReferenceArray`.

Overview#

The official documentation of the `AtomicReferenceArray` states that it is an array of object references, that each one, may be updated atomically. However, it is easy to get confused with all the other array constructs and `AtomicReferenceArray`, so we'll go through one by one to clarify the differences. Also, according to the Java specification, assignments in Java except for primitive types `long` and `double` are atomic, which may confuse the reader as to what does *atomically updating* an element of an `AtomicReferenceArray` instance means?

Remember to draw a distinction between assignment and update of an atomic variable. Assignment involves changing the reference (memory location) pointed to by an atomic variable e.g.

```
// first assignment
AtomicReference<Object> atomicReference = new AtomicReference<>(nu
ll);

// assigning a different AtomicReference object
atomicReference = new AtomicReference<>(Long.class);
```

Updating atomically, implies that the value of an atomic variable holds is changed based on an expected value the variable currently holds e.g.

```
// first assignment
AtomicReference<Object> atomicReference = new AtomicReference<>(nu
ll);

// atomic update
atomicReference.compareAndSet(null, Long.class);
```

The atomic update has consequences for memory visibility and establishes the happens-before relationship as we'll learn shortly. In contrast, an assignment operation may not establish a *happens-before* relationship if the variable isn't marked `volatile`.

Array Reference#

First up, we'll discuss what is an array reference. There is a distinction between an array reference and the elements of the array. Array is a sequence of values and the values are the elements of the array, all of the same type. The array is created using the `new` operator as shown below:

```
Long[] myLongs = new Long[10];
```

The `new` operator, allocates memory on the heap for the array and returns the address of the memory allocation into the variable `myLongs`. The variable `myLongs` isn't the array; rather the variable holds the address of the memory location where the elements of the array live. The memory location's address is stored in the variable `myLongs` and the variable `myLongs` is said to be a reference to the array.

This distinction between array reference and the array is important because one could update the array reference in a thread-safe manner but the elements of the array may still be updated in a thread-unsafe manner. Consider the following methods:

```
Long[] myLongs = new Long[10];

// method is thread-safe
void increaseArraySize() {
    synchronized (this) {
        Long[] newArray = new Long[myLongs.length * 2];

        // copy old array elements into new array
        for (int i = 0; i < myLongs.length; i++)
            newArray[i] = myLongs[i];

        // update the reference
        myLongs = newArray;
    }
}

// method isn't thread-safe
```

```
void set(int index, Long newVal) {  
    myLongs[index] = newVal;  
}
```

Note that the array reference is being manipulated in a thread-safe manner but the elements of the array are not.

Difference between array and volatile array#

Another source of confusion is the following snippet:

```
Long[] myLongs = new Long[10];  
volatile Long[] volatileMyLongs = new Long[10];
```

The reference `myLongs` is prone to getting cached by a processor whereas the reference `volatileMyLongs` is always read and written from main memory. However, the elements for both the arrays can get cached by a processor and multiple threads working on the array without appropriate synchronization constructs can see stale values for array elements in addition to thread-safety issues. Going back to our example, let's say we modify the `increaseArraySize()` method as follows:

```
void increaseArraySize() {  
    Long[] newArray = new Long[myLongs.length * 2];  
  
    // copy old array elements into new array  
    for (int i = 0; i < myLongs.length; i++)  
        newArray[i] = myLongs[i];  
  
    // update the reference  
    myLongs = newArray;  
}
```

If only the main thread ever invokes the `increaseArraySize()` method, then to avoid the locking overhead, we may remove the `synchronized` block from the method. However, without marking `newLongs` as `volatile`, it could happen that the main thread creates a new array and updates the address in the local cache and never in the main memory. Consequently, other threads never see the new array and keep working on the old one. It could also happen that other threads have the reference cached and never retrieve the latest reference from the main memory since `myLongs` isn't marked volatile. Either way, marking the reference i.e. the variable `myLongs` volatile solves the memory visibility issue. However, marking `myLongs` as `volatile` has no effect on the array elements. Consider a thread that executes the following snippet:

```
Long cachedObject = myLongs[0];
```

The `Long` object at zero-th index isn't `volatile` and will suffer from memory visibility issues caused by caching, even though the reference to the array has been marked `volatile`.

Also, bear in mind that `volatileMyLongs` reference is volatile, and `volatileMyLongs` is NOT an array of volatile `Long` objects.

Difference between volatile array

and `AtomicReferenceArray`#

The next question we'll address is the difference among the three statements in the following snippet:

```
volatile Long[] volatileMyLongs = new Long[10];
AtomicReferenceArray<Long> atomicReferenceArray = new AtomicReferenceArray
<>(10);
volatile AtomicReferenceArray volatileAtomicReferenceArray = new AtomicRef
erenceArray(10);
```

The reference `volatileMyLongs` is volatile but the reference `atomicReferenceArray`, which points to an object of type `AtomicReferenceArray` on the heap isn't. The elements of both the arrays can be updated atomically, since they are references and the Java specification mandates atomic assignments for references. However, the array elements of `volatileMyLongs` can suffer from memory visibility issues such as being cached in the processor's local memory and any updates to them will not establish the *happens-before* relationship with other variables observable to a thread. In the case of the `atomicReferenceArray` variable, the individual elements at each index are always updated in the main memory and a stale value is never observed by any thread. Furthermore, manipulating a single element of `atomicReferenceArray` also establishes the *happens-before* relationship with other variables in the scope.

The variable `volatileAtomicReferenceArray` reference, itself and its array elements don't suffer from memory visibility issues. If the reference is updated after initialization using the following snippet, the change will be visible to all threads:

```
volatileAtomicReferenceArray = newReference;
```

A similar update as above to `atomicReferenceArray` variable can potentially be cached and not observable by all threads without using synchronization constructs.

Difference between `AtomicReferenceArray` and array of `AtomicReference`-s#

The astute reader would question the difference between instantiating an array of `AtomicReference`-s vs instantiating an object of `AtomicReferenceArray`. Consider the following snippet:

```
AtomicReferenceArray<Long> atomicReferenceArray = new AtomicReferenceArray<Long>(100);
AtomicReference[] arrayOfAtomicReferences = new AtomicReference[100];
```

Functionally, the above two are equivalent, however, the instance of `AtomicReferenceArray` occupies less memory than an array of `AtomicReference`-s of equivalent size. The memory saving can become significant when the size of the array is in the thousands or millions.

Summary of differences#

Snippet	Is reference assignment atomic?	Is update atomic?	Reference update establishes <i>happens-before</i> relationship?	Element update establishes <i>happens-before</i> relationship
<code>Long[] myLongs = new Long[10];</code>	Yes	No	No	No
<code>volatile Long[] volatileMyLongs = new Long[10];</code>	Yes	No	Yes	No
<code>AtomicReferenceArray<Long> atomicReferenceArray;</code>	Yes	Yes	No	Yes
<code>volatile AtomicReferenceArray<Long> volatileAtomicReferenceArray;</code>	Yes	Yes	Yes	Yes

Example#

In the example below, we create an instance of `AtomicReferenceArray<Long>` typed on the `Long` class. Next, we run fifteen threads and each one of them attempts to initialize each element of the array with a `Long` object. The reference of the `Long` object is stored in the array element. No matter how many times we run the program, each element of the array should only be initialized by one thread.

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

class Demonstration {

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

        int arrayLength = 10;
        AtomicReferenceArray<Long> atomicReferenceArray = new
        AtomicReferenceArray<>(arrayLength);
        ExecutorService executor = Executors.newFixedThreadPool(15);

        try {

            for (int i = 0; i < arrayLength; i++) {

                executor.submit(new Runnable() {
                    @Override
                    public void run() {

                        for (int index = 0; index < arrayLength; index++) {

                            Long currentVal = atomicReferenceArray.get(index);

                            // attempt to initialize
                            if (currentVal == null && atomicReferenceArray.compareAndSet(index,
                                currentVal, new Long(index))) {
                                System.out.println(Thread.currentThread().getName() + " initialized index " +
                                index);
                            }
                        }
                    }
                });
            }

        } finally {
            executor.shutdown();
            executor.awaitTermination(1L, TimeUnit.HOURS);
        }
    }
}
```

```
    }
}
}
```

AtomicStampedReference

Learn how the `AtomicStampedReference` class can be used to address the ABA problem that can manifest when classes manage their own memory.

Overview#

Similar to the `AtomicReference` class, the `AtomicStampedReference` class contains a reference to an object and additionally an integer stamp. The integer stamp is usually incremented whenever the object pointed to by the reference is modified. The `AtomicStampedReference` class can be thought of as a generalization of the `AtomicMarkableReference` class, replacing the boolean with an integer. The integer stamp stored alongside the reference can also be used for other purposes such as maintaining one of the finite states a data structure can be in . The stamp and reference fields can be updated atomically, either together or individually.

The code widget below demonstrates the use of `AtomicStampedReference` and its common APIs.

```
import java.util.concurrent.atomic.AtomicStampedReference;

class Demonstration {

    public static void main( String args[] ) {

        Long myLong = new Long(3);
        Long anotherLong = new Long(7);

        // set the initial stamp to 1 and reference to myLong
        AtomicStampedReference<Long> atomicStampedReference = new
        AtomicStampedReference<>(myLong,1);

        // we attempt to change the object reference but use the incorrect stamp and the
        // compareAndSet fails
        boolean result = atomicStampedReference.compareAndSet(myLong, anotherLong, 0, 1);
        System.out.println("Was compareAndSet() successful : " + result + " and object value is " +
        atomicStampedReference.getReference().toString());

        // we attempt compareAndSet again with the right expected stamp
        result = atomicStampedReference.compareAndSet(myLong, anotherLong, 1, 2);
```

```

        System.out.println("Was compareAndSet() successful : " + result + " and object value is " +
atomicStampedReference.getReference().toString());

        // Retrieve the current stamp and reference using the get() method
        int[] currStamp = new int[1];
        Long reference = atomicStampedReference.get(currStamp);
        System.out.println("current stamp " + currStamp[0] + " reference value " +
reference.toString());
    }
}

```

Note that the current reference and stamp can be retrieved using the `get(int[] stampHolder)` call. The reference is the return value from the method invocation and the passed-in array is populated with the current stamp. The `AtomicStampedReference` solves the ABA problem that can occur when using the plain `AtomicReference` class. We'll discuss them next.

ABA problem#

The ABA problem is described in detail in the Atomic lesson of the course. But briefly, algorithms that dynamically manage their own memory and use conditional synchronization operations such as `compareAndSet()` can run into this issue. Usually, the pattern involves the `compareAndSet()` method about to modify a reference from say *A* to some other reference but before `compareAndSet()` gets a chance to execute, another thread modifies *A* to *B* and then back to *A*. The `compareAndSet()` proceeds forward and succeeds when its effect on the data structure is not what was initially desired. We'll see a concrete set-up of the ABA problem next.

Example#

Consider the following example where the `Demonstration` class maintains a concurrent list of free `Node`-s that allows it to recycle nodes and manage its own memory. This is an instructional example to demonstrate the ABA problem and lacks purpose, but the described scenario is applicable to data structures that may choose to conduct memory management on their own for efficiency purposes. In this example, threads either append to or remove the head of a `LinkedList` of sorts. The setup is as follows:

1. The `head` of the list is stored as an `AtomicReference` and threads can update the head using compare and set API.
2. At the start of the program, the main thread appends ten nodes to the list.

3. Thread1 starts and prepares to dequeue the current `head`. In doing so it has to set the `head` variable to the next of the current head. But just before invoking `compareAndSet()`, we sleep Thread1 to simulate Thread1 being context-switched. Note that at this point, Thread1 has read the current head and the next of current head in local variables.
4. Thread2 starts and immediately sleeps to allow Thread1 to hit its sleep statement and complete the previous step.
5. Thread2 proceeds to dequeue five nodes and places them in the list of free nodes. Including the head and its next node that was read-in by Thread1 before Thread1 went to sleep.
6. Thread2 now appends a new value to the list but it reuses the node from the free list. The reused Node carries a different value than before but it is the same head value that was read-in by Thread1
7. Thread1 wakes-up and proceeds to execute `compareAndSet()` and since the current `head` is the same reference as the one read-in by Thread1 in step number 3, the `compareAndSet()` succeeds when it should fail. Also the new `head` is set to node that is in the free nodes list. Note that when Thread1 read the head value its value was 10 and when Thread1 executed `compareAndSet()` its value was 99.

The complete example with comments appears in the widget below. Note that we have used `Thread.sleep()` for manifesting the ABA problem, but in practise `Thread.sleep()` should never be used in production code for synchronization among threads.

Main.java

```
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.atomic.AtomicReference;

class Demonstration {

    private static ConcurrentLinkedQueue<Node> availableNodes = new
    ConcurrentLinkedQueue<>();
    private static AtomicReference<Node> head = new AtomicReference<>(null);

    public static void main( String args[] ) throws Exception {
        Node currHead = null;
        Node node = null;

        // nodes are inserted by the main thread with values
        // ranging from 0 to 9
        for (int i = 0; i < 10; i++) {
            node = new Node(i);
            availableNodes.add(node);
        }

        // Thread 1 starts here
        currHead = head.get();
        Node nextNode = currHead.getNext();
        head.set(nextNode);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Thread 1: Head is " + currHead.getValue());
        System.out.println("Thread 1: Next is " + nextNode.getValue());

        // Thread 2 starts here
        currHead = head.get();
        Node previousNode = currHead.getPrevious();
        head.set(nextNode);
        previousNode.setNext(nextNode);
        System.out.println("Thread 2: Head is " + currHead.getValue());
        System.out.println("Thread 2: Previous is " + previousNode.getValue());
        System.out.println("Thread 2: Next is " + nextNode.getValue());
    }
}
```

```

        node.next = currHead;
        head.compareAndSet(currHead, node);
        currHead = node;
    }

    System.out.println("Initial list : ");
    printNodes();

    // creating Thread1
    Thread thread1 = new Thread(new Runnable() {
        @Override
        public void run() {

            // Thread1 reads-in the current head and its next node
            Node currentHead = head.get();
            Node nextHead = currentHead.next;

            System.out.println("Thread 1 sees head = " + currentHead.val + " and head.next = " +
nextHead.val);

            // sleep Thread1
            try {
                Thread.sleep(5000);
            } catch (InterruptedException ie) {
                // ignore
            }

            System.out.println("Thread1 about to compare and set");

            if (head.compareAndSet(currentHead, nextHead)) {
                System.out.println("Thread1 successfully updated head. List looks as follows: ");
                printNodes();
            } else {
                System.out.println("CAS failed in Thread1");
            }
        }
    });

    thread1.start();

    // set-up Thread2
    Thread thread2 = new Thread(new Runnable() {
        @Override
        public void run() {

            // wait for Thread 1 to reach its sleep statement
            try {
                Thread.sleep(1000);

```

```

} catch (InterruptedException ie) {
    // ignore
}

// dequeue first five nodes from the list and place them
// in the free nodes list
Node currHead = null;
for (int i = 0; i < 5; i++) {
    currHead = head.get();
    head.compareAndSet(currHead, currHead.next);
    currHead.val = -1; // set to -1 to denote the node is in recycle list
    currHead.next = null;
    availableNodes.add(currHead);
}

currHead = head.get();
Node newHead = availableNodes.remove();
newHead.val = 99; // set a new value
newHead.next = currHead;
if (head.compareAndSet(currHead, newHead)) {
    System.out.println("Thread 2 successfully updates. List is as follows : ");
    printNodes();
}
}
});

thread2.start();

// wait for threads to exit
thread1.join();
thread2.join();
}

// helper method to print the list
static void printNodes() {
    Node currHead = head.get();
    boolean start = true;
    while (currHead != null) {
        if (start) {
            start = false;
            System.out.print(currHead.val + " (head) -> ");
        } else {
            System.out.print(currHead.val + " -> ");
        }
        currHead = currHead.next;
    }
    System.out.println();
}
}

```

```
}
```

Node.java

```
public class Node {  
    public Node(int val) {  
        this.val = val;  
    }  
  
    int val;  
    Node next;  
}
```

The above result is very interesting, note that the entire code snippet is thread-safe since the operations on the list are always done using `compareAndSet()`, but because of ABA the snippet brings the data structure into a state where the head is set to a reference pointing to a node in the recycle node list.

Fixing the ABA problem

with `AtomicStampedReference`#

The above example is fixed using the `AtomicStampedReference` class which uses an integer stamp alongside to detect if a node was reused. The same scenario from the previous widget is repeated and the `compareAndSet()` in Thread1 correctly fails.

Main.java

```
import java.util.concurrent.ConcurrentLinkedQueue;  
import java.util.concurrent.atomic.AtomicStampedReference;  
  
class Demonstration {  
  
    private static ConcurrentLinkedQueue<Node> availableNodes = new  
    ConcurrentLinkedQueue<>();  
    private static AtomicStampedReference<Node> head = new  
    AtomicStampedReference<>(null, 1);  
  
    public static void main( String args[] ) throws Exception {  
  
        Node currHead = null;  
        Node node = null;  
  
        // nodes are inserted by the main thread with values  
        // ranging from 0 to 9
```

```

for (int i = 0; i < 10; i++) {
    node = new Node(i);
    node.next = currHead;
    int currStamp = head.getStamp();
    head.compareAndSet(currHead, node, currStamp, currStamp + 1);
    currHead = node;
}

System.out.println("Initial list : ");
printNodes();

// creating Thread1
Thread thread1 = new Thread(new Runnable() {
    @Override
    public void run() {

        // Thread1 reads-in the current head and its next node
        Node currentHead = head.getReference();
        int currHeadStamp = head.getStamp();
        Node nextHead = currentHead.next;

        System.out.println("Thread 1 sees head (with stamp " + currHeadStamp + ") = " +
currentHead.val + " and head.next = " + nextHead.val);

        // sleep Thread1
        try {
            Thread.sleep(5000);
        } catch (InterruptedException ie) {
            // ignore
        }
    }

    System.out.println("Thread1 about to compare and set");

    if (head.compareAndSet(currentHead, nextHead, currHeadStamp, currHeadStamp +
1)) {
        System.out.println("Thread1 successfully updated head. List looks as follows: ");
        printNodes();
    } else {
        System.out.println("CAS failed in Thread1");
    }
}
});

thread1.start();

// set-up Thread2
Thread thread2 = new Thread(new Runnable() {
    @Override

```

```

public void run() {

    // wait for Thread 1 to reach its sleep statement
    try {
        Thread.sleep(1000);
    } catch (InterruptedException ie) {
        // ignore
    }

    // dequeue first five nodes from the list and place them
    // in the free nodes list
    Node currHead = null;
    int currStamp;
    for (int i = 0; i < 5; i++) {
        currHead = head.getReference();
        currStamp = head.getStamp();
        head.compareAndSet(currHead, currHead.next, currStamp, currStamp + 1);
        currHead.val = -1; // set to -1 to denote the node is in recycle list
        currHead.next = null;
        availableNodes.add(currHead);
    }

    currHead = head.getReference();
    currStamp = head.getStamp();
    Node newHead = availableNodes.remove();
    newHead.val = 99; // set a new value
    newHead.next = currHead;
    if (head.compareAndSet(currHead, newHead, currStamp, currStamp + 1)) {
        System.out.println("Thread 2 successfully updates. List is as follows : ");
        printNodes();
    }
}
});

thread2.start();

// wait for threads to exit
thread1.join();
thread2.join();

System.out.println("List when main program exits is as follows ");
printNodes();

}

// helper method to print the list
static void printNodes() {
    Node currHead = head.getReference();

```

```

int currStamp = head.getStamp();
boolean start = true;
while (currHead != null) {
    if (start) {
        start = false;
        System.out.print(currHead.val + " (head with stamp " + currStamp + ") -> ");
    } else {
        System.out.print(currHead.val + " -> ");
    }
    currHead = currHead.next;
}
System.out.println();
}
}

```

Node.java

```

public class Node {
    public Node(int val) {
        this.val = val;
    }

    int val;
    Node next;
}

```

The above code repeats the same sequence of events as in the previous widget but because we switched to `AtomicStampedReference` class for storing the head of the queue, the `compareAndSet()` call fails in Thread1 and the list remains unmodified from what it looked like after Thread2's operations, when the program exits.

AtomicMarkableReference

Learn to use the `AtomicMarkableReference` class for designing lock-free data structure such as linked-lists.

Overview#

The `AtomicMarkedReference` class is similar to the `AtomicStampedReference` class in that it holds a reference to an object but instead of an integer stamp it takes in a boolean value, called the mark. Both these fields can be updated atomically either together or individually. One could argue that `AtomicStampedReference` class would behave similarly

to `AtomicMarkableReference` class if it accepted only two possible values for the integer stamp argument.

The code widget below demonstrates some of the basic operations when working with the `AtomicMarkableReference` class.

```
import java.util.concurrent.atomic.AtomicMarkableReference;

class Demonstration {

    public static void main( String args[] ) {

        Long myLong = new Long(5);
        Long anotherLong = new Long(7);
        AtomicMarkableReference<Long> atomicMarkableReference = new
        AtomicMarkableReference<>(myLong, false);

        // attempt to change the long value with the wrong expected mark
        boolean wasSuccess = atomicMarkableReference.compareAndSet(myLong, anotherLong,
        true, true);
        System.out.println("compare and set succeeded " + wasSuccess + " current value " +
        atomicMarkableReference.getReference());

        // attempt to change the long value with the right expected mark
        wasSuccess = atomicMarkableReference.compareAndSet(myLong, anotherLong, false,
        true);
        System.out.println("compare and set succeeded " + wasSuccess + " current value " +
        atomicMarkableReference.getReference());
    }
}
```

Example#

`AtomicMarkableReference` can be put to good use when designing lock-free lists, such as a set representation backed by a linked-list. In this lesson, we'll not go through the implementation of a lock-free list but discuss parts of the implementation where the class `AtomicMarkableReference` is applicable.

Consider the `SimpleNode` class which represents a node in the linked list. The class uses an `AtomicReference` for the `next` node in the list, so that it can be updated atomically. The goal is to keep the list lock-free and update the `next` of the node correctly when a node is deleted. We'll create a set-up that creates a race condition and causes the list to reach an inconsistent state.

Say we have three nodes linked as follows:

NodeA → NodeB → NodeC → null

Say we have two threads `thread1` and `thread2`, each one trying to delete NodeA and NodeB respectively. The following sequence of events can cause the NodeC to not be deleted:

1. `thread1` stores the `next` of NodeA, which points to NodeB and the `next` of NodeB which points to NodeC in the variables `expected` and `newNext` respectively. These variables serve as the expected and the new value for the `compareAndSet()` method of the `AtomicReference`.
2. At this point `thread1` is suspended. We simulate that by sleeping the thread.
3. `thread2` starts and attempts to delete NodeC. `thread1` computes the expected value for the `next` of NodeB, which is NodeC itself and the new next value which should be null, since NodeC is the last node of the list.
4. `thread2` continues and successfully deletes NodeC.
5. `thread1` wakes up and proceeds to execute the `compareAndSet()` call, which should succeed because NodeA still points to NodeB (the expected value) but the new `next` points to NodeC which has been deleted.
6. The program ends in a state where NodeC is not deleted and remains part of the list.

The above scenario is depicted in the code widget below:

Main.java

```
class Demonstration {  
  
    public static void main( String args[] ) throws Exception {  
        SimpleNode nodeC = new SimpleNode(3, null);  
        SimpleNode nodeB = new SimpleNode(2, nodeC);  
        SimpleNode nodeA = new SimpleNode(1, nodeB);  
  
        // NodeA --> NodeB --> NodeC  
  
        Thread thread1 = new Thread(new Runnable() {  
            @Override  
            public void run() {  
  
                // deleting NodeB
```

```

SimpleNode expected = nodeA.next.get();
SimpleNode next = nodeB.next.get();

// thread goes to sleep
try {
    Thread.sleep(3000);
} catch (InterruptedException ie) {
    // ignore
}

// thread wakes-up and successfully updates referece
nodeA.next.compareAndSet(expected, next);
}

});

thread1.start();
Thread.sleep(2000);

Thread thread2 = new Thread(new Runnable() {
    @Override
    public void run() {
        // deleting NodeC
        SimpleNode expected = nodeB.next.get();
        SimpleNode next = nodeC.next.get();
        nodeB.next.compareAndSet(expected, next);
    }
});

thread2.start();

thread1.join();
thread2.join();

SimpleNode start = nodeA;
while (start != null) {
    System.out.println(start.value + " ");
    start = start.next.get();
}
}
}
}

```

SimpleNode.java

```

import java.util.concurrent.atomic.AtomicReference;

public class SimpleNode {
    int value;

```

```

AtomicReference<SimpleNode> next;

public SimpleNode(int value, SimpleNode next) {
    this.value = value;
    this.next = new AtomicReference<>(next);
}
}

```

The reason we end up in an inconsistent state is because we have no way of tracking the state of a given node if it has already been deleted or not. In our example, NodeC was deleted but `thread1` didn't realize that NodeC was not part of the list anymore. One possible solution is to store a boolean alongside each node that indicates the deletion status of the node. If set to true, the node is considered deleted but may not have been removed from the list yet.

This functionality is provided by the `AtomicMarkableReference` class. We can use `AtomicMarkableReference` to store the `next` for each node. Whenever, updating the `next` field of a node using the `compareAndSet()` method, we'll pass in two expected values instead of one. The first expected value will be the reference of the object and the second a boolean value. When we update the `next` field of a node, we provide the new reference and `false` for the boolean value, implying that we only update the reference, if the node itself hasn't been marked for deletion. In case, the delete operation finds that the predecessor of the node being deleted is already marked for deletion then the current delete operation fails physically, in the sense that the reference in the `next` field of the predecessor isn't updated but the node being deleted has its mark set to true to indicate a logical delete.

If we re-run the same example with our modified algorithm, `thread1` will succeed to update the reference after resumption from sleep, however, `thread2` will not be able to delete nodeC. The list would still comprise of NodeA and NodeC but not be in an inconsistent state because the mark for NodeC will be true, indicating that the node has been logically deleted. The operation to delete NodeC can be either re-tried immediately or lazily deleted at a later time when another operation traverses the list and removes all the nodes marked true.

Main.java

```

class Demostration {
    public static void main( String args[] ) throws Exception {
        Node nodeC = new Node(3, null);
        Node nodeB = new Node(2, nodeC);
        Node nodeA = new Node(1, nodeB);
    }
}

```

```

// nodeA --> nodeB --> nodeC

Thread thread1 = new Thread(new Runnable() {
    @Override
    public void run() {

        // deleting NodeB
        // mark NodeB as deleted
        nodeB.next.set(nodeB.next.getReference(), true);
        Node expected = nodeA.next.getReference();
        Node next = nodeB.next.getReference();

        // thread goes to sleep
        try {
            Thread.sleep(3000);
        } catch (InterruptedException ie) {
            // ignore
        }

        // thread wakes-up and attempts to compareAndSet.
        nodeA.next.compareAndSet(expected, next, false, false);
    }
});

thread1.start();
Thread.sleep(2000);

Thread thread2 = new Thread(new Runnable() {
    @Override
    public void run() {
        // deleting NodeC
        nodeC.next.set(nodeC.next.getReference(), true);
        Node expected = nodeB.next.getReference();
        Node next = nodeC.next.getReference();
        // The result of deleting nodeC is false.
        nodeB.next.compareAndSet(expected, next, false, false);
    }
});

thread2.start();

thread1.join();
thread2.join();

Node start = nodeA;
while (start != null) {
    boolean[] mark = new boolean[1];

```

```

        start.next.get(mark);
        System.out.println(start.value + " (mark : " + mark[0] + ")");
        start = start.next.getReference();
    }
}
}
}

```

Node.java

```

import java.util.concurrent.atomic.AtomicMarkableReference;

public class Node {
    int value;
    AtomicMarkableReference<Node> next;

    public Node(int value, Node next) {
        this.value = value;
        // initially marked the Node as false
        this.next = new AtomicMarkableReference<Node>(next, false);
    }
}

```

This example captures the gist of the utility of the `AtomicMarkableReference` class and demonstrates how additional state about a data structure can be maintained.

Exchanger

Guide to using the `Exchanger<V>` class.

Overview#

As the name indicates, `Exchanger` is a construct that can be used to make bidirectional transfers of objects between two threads. Each thread invokes the `exchange()` method with an item the thread wants to exchange with the other thread. A thread blocks when making the `exchange()` call until the other thread invokes the `exchange()` method.

In case of multiple threads, it is not possible for a thread to choose its partner to exchange objects with. Note that the `exchange()` must be invoked an even number of times during the course of a program to ensure no thread remains blocked when the program exits.

An **Exchanger** may be viewed as a bidirectional form of a **SynchronousQueue**. Exchangers may be useful in applications such as genetic algorithms and pipeline designs.

Example#

In the simple example below, we have two threads that exchange **String** objects with each other. Each thread shares its name with the other thread. The output of the program shows the string that each thread receives from the other thread.

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

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        // create an exchanger object
        Exchanger<String> exchanger = new Exchanger<>();

        // create an executor service
        ExecutorService executorService = Executors.newFixedThreadPool(5);

        try {
            // submit the first task
            executorService.submit(new Runnable() {
                @Override
                public void run() {

                    String name = Thread.currentThread().getName();
                    String received = "";
                    try {
                        received = exchanger.exchange(name);
                    } catch (InterruptedException ie) {

                    }

                    System.out.println("I am thread " + name + " and I received the string " + received);
                }
            });

            // submit the second task
            executorService.submit(new Runnable() {
                @Override
                public void run() {

                    String name = Thread.currentThread().getName();
                    String received = "";
                    try {

                    }
                }
            });
        }
    }
}
```

```
        received = exchanger.exchange(name);
    } catch (InterruptedException ie) {
        }
        System.out.println("I am thread " + name + " and I received the string " + received);
    }
});
} finally {
    // remember to shutdown the executor service
    executorService.shutdown();
    executorService.awaitTermination(1, TimeUnit.HOURS);
}
}
```

In the second example, we have ten threads exchange their names. The very first thread blocks on the invocation to `exchange()`. Next, the second thread comes along and is paired with the first blocked thread and the two exchange strings and move on. This pattern continues with the third thread being paired with the fourth, the fifth with the sixth, so on and so forth.

Note that we can't have the first thread to pair with, say the seventh thread, i.e. the developer can't influence how threads are paired.

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

class Demonstration {

    static Exchanger<String> exchanger = new Exchanger<>();

    public static void main( String args[] ) throws InterruptedException {
        ExecutorService executorService = Executors.newFixedThreadPool(10);

        try {
            for (int i = 0; i < 10; i++) {
                executorService.submit(new Runnable() {
                    @Override
                    public void run() {
                        work();
                    }
                });
            }
        } finally {
            executorService.shutdown();
            executorService.awaitTermination(1, TimeUnit.HOURS);
        }
    }
}
```

```

}

static void work() {
    String name = Thread.currentThread().getName();
    String received = "";

    try {
        received = exchanger.exchange(name);
    } catch (InterruptedException ie) {
        // ignore for now
    }

    System.out.println("I am thread " + name + " and I received the string " + received);
}

}

```

The output of the above widget shows which thread gets paired with which other thread for the exchange of objects.

Phaser

Comprehensive guide with executable examples to using Phaser, an advanced and sophisticated synchronization barrier construct.

Overview#

The `Phaser` class is an extension of the functionality offered by `CyclicBarrier` and `CountDownLatch` classes and is more flexible in use. One stark difference is that the `Phaser` class allows the number of registered parties that synchronize on a phaser to vary over time. The `Phaser` can be repeatedly awaited similar to a `CyclicBarrier`.

Example#

Apart from specifying the number of threads/tasks to synchronize in the constructor, threads/tasks can also register with an instance of `Phaser` using the `register()` or the `bulkRegister(int)` methods. Note, that if a thread `register()`-s with an instance of `Phaser` there's no way for the thread to query the instance to determine if it registered with the instance, i.e. there's no internal book-keeping maintained by the `Phaser` instance. However, if such behavior is desired the `Phaser` class can be subclassed and the book-keeping functionality added.

The program below exercises some of the APIs exposed by [Phaser](#) to register threads with the barrier. Run the program and study the comments before we discuss them.

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

class Demonstration {
    public static void main( String args[] ) throws Exception {
        // create an executor service
        ExecutorService executorService = Executors.newFixedThreadPool(5);

        // create an instance of Phaser class and register only a single that will arrive
        // at the barrier
        Phaser phaser = new Phaser(1);

        try {
            // a thread registers with the Phaser post construction of the instance
            executorService.submit(new Runnable() {
                @Override
                public void run() {
                    phaser.register();
                }
            });
        }

        // main thread bulk-registers two more parties
        phaser.bulkRegister(2);

        // main thread registering one more party.
        phaser.register();

        // we now have 5 parties registered with the Phaser instance
        // we instantiate four threads and have them arrive at the barrier
        for (int i = 0; i < 4; i++) {
            executorService.submit(new Runnable() {
                @Override
                public void run() {
                    phaser.arriveAndAwaitAdvance();
                    System.out.println(Thread.currentThread().getName() + " moving past barrier.");
                }
            });
        }

        // sleep for a while so that previous threads can arrive at the barrier
        Thread.sleep(2000);

        // before arriving at the barrier, print the counts of parties
        System.out.println(Thread.currentThread().getName() + " just before arrived. \n Arrived
parties: " + phaser.getArrivedParties() +
```

```

        "\n Registered parties: " + phaser.getRegisteredParties() +
        "\n Unarrived parties: " + phaser.getUnarrivedParties());
    phaser.arriveAndAwaitAdvance();

} finally {
    // remember to shutdown the executor in a finally block
    executorService.shutdown();
    executorService.awaitTermination(1, TimeUnit.HOURS);
}

// main thread prints party counts for the barrier
System.out.println(Thread.currentThread().getName() + " exiting. \n Arrived parties: " +
phaser.getArrivedParties() +
        "\n Registered parties: " + phaser.getRegisteredParties() +
        "\n Unarrived parties: " + phaser.getUnarrivedParties());
}
}

```

Notice that the main thread is responsible for registering 3 parties with the `Phaser` instance after the instance has been constructed but arrives at the barrier only once, i.e. it is not necessary that the thread that invokes `register()` must also be the same thread that arrives at the barrier.

Arriving and Deregistering#

Consider a scenario where we want all the spawned threads/tasks to wait until the main thread has finished initialization or performed some tasks before we want the spawned threads to proceed. We could initialize the `Phaser` with a count of one more than the number of threads we plan to spawn, and then have the main thread do the required work. Finally, the main thread arrives at and deregisters with the barrier at the same time. This releases the spawned threads that have already been waiting at the barrier and reduces the number of parties required to synchronize at the barrier by one for future. The described example appears in the program below.

```

import java.util.concurrent.*;

class Demonstration {
    public static void main( String args[] ) throws Exception {
        // create an executor service
        ExecutorService executorService = Executors.newFixedThreadPool(15);

        // create an instance of Phaser with 3 registered parties
        Phaser phaser = new Phaser(3);

        try {

```

```

        for (int i = 0; i < 2; i++) {
            executorService.execute(new Runnable() {
                @Override
                public void run() {
                    System.out.println(Thread.currentThread().getName() + " about to arrive at the
barrier");
                    phaser.arriveAndAwaitAdvance();
                    System.out.println("Thread " + Thread.currentThread().getName() + " moving past
the phaser once");
                    phaser.arriveAndAwaitAdvance();
                    System.out.println(Thread.currentThread().getName() + " moving past the phaser
twice");
                }
            });
        }

        // sleep for a while to simulate work that the main thread needs to get done before
        // letting the spawn threads proceed forward.
        Thread.sleep(5000);

        phaser.arriveAndDeregister();
        System.out.println(Thread.currentThread().getName() + " past the barrier. \n Arrived
parties: " + phaser.getArrivedParties() +
        "\n Registered parties: " + phaser.getRegisteredParties() +
        "\n Unarrived parties: " + phaser.getUnarrivedParties());

    } finally {
        // remember to shutdown the barrier in a finally block
        executorService.shutdown();

        // wait for spawned threads to finish
        executorService.awaitTermination(1, TimeUnit.HOURS);
    }

    System.out.println("Program exiting");
}
}

```

From the output of the program note that once the main thread is past the barrier the number of registered parties with the barrier is reduced by one than when we initially created the barrier.

Non-blocking arrival#

In the previous example we used the blocking method `arriveAndAwaitAdvance()` to synchronize at the barrier. However, there may be scenarios where we want a thread/task to record arrival at a

barrier but not block. For such use cases **Phaser** class provides two non-blocking methods:

- `arrive()`
- `arriveAndDeregister()`

Here's an example program that demonstrates the use of `arrive()`. Note that the barrier is initialized with a party count of 5 and only the main thread records arrival and moves past the barrier without blocking.

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

class Demonstration {

    public static void main( String args[] ) {
        // create an instance of Phaser class and register 5 parties
        Phaser phaser = new Phaser(5);

        // main thread records arrival at the barrier
        phaser.arrive();

        System.out.println("main thread moving past the barrier. \n Unarrived parties " +
        phaser.getUnarrivedParties());
    }
}
```

Phases of a Phaser#

You might be wondering why the **Phaser** class is named **Phaser** and it is because an instance of the class moves from one phase to another as the registered parties record arrival and advance. The starting phase is numbered 0, when all the registered parties arrive at the barrier, the **Phaser** instance advances to the next phase which is 1. This pattern continues until the phase reaches the maximum allowed `Integer.MAX_VALUE` and then wraps to zero. The phase numbered 0 is never arrived at by synchronizing parties. The synchronization methods return the arrival phase which starts at 1. The program below prints the phases of a **Phaser** instance as the main thread records arrival and then advances to the next phase.

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

class Demonstration {

    public static void main( String args[] ) {
        // create an instance of Phaser class and register a single party
        Phaser phaser = new Phaser(1);
```

```

// get the initial phase number
int phase = phaser.getPhase();
System.out.println("starting at phase : " + phase);

// arrive and print the current phase
phase = phaser.arriveAndAwaitAdvance();
System.out.println("phase " + phase);

// arrive and print the current phase
phase = phaser.arriveAndAwaitAdvance();
System.out.println("phase " + phase);

// arrive and print the current phase
phase = phaser.arriveAndAwaitAdvance();
System.out.println("phase " + phase);

}

}

```

We have another example demonstrating the main thread waiting until 10 phases of a **Phaser** instance complete and then advancing forward. You can consider a scenario where we want the main thread to proceed forward after worker threads have synchronized a certain number of times over a barrier. The example with code comments appears below:

```

import java.util.concurrent.*;

class Demonstration {
    public static void main( String args[] ) throws Exception {
        // create an executor service
        ExecutorService executorService = Executors.newFixedThreadPool(15);

        Phaser phaser = new Phaser(2);

        try {
            // register main thread with the phaser
            int arrivalPhase = phaser.register();

            // simulate work by two threads that synchronize 10 times at the barrier
            for (int i = 0; i < 2; i++) {
                executorService.submit(new Runnable() {
                    @Override
                    public void run() {
                        for (int i = 0; i < 15; i++) {
                            phaser.arriveAndAwaitAdvance();

                            // simulate worker threads execute some other tasks after 10 iterations
                            if (i > 10) {
                                try {

```

```

        Thread.sleep(1000);
    } catch (InterruptedException ie) {
        // ignore for now
    }
}
System.out.println(Thread.currentThread().getName() + " proceeding forward.");
}
});
}

while (arrivalPhase < 10) {
    arrivalPhase = phaser.arriveAndAwaitAdvance();
    System.out.println("main thread arrived at phase " + arrivalPhase);
}

// non-blocking call
phaser.arriveAndDeregister();
System.out.println("main thread past the barrier");
} finally {
    // remember to shutdown the executor
    executorService.shutdown();
    executorService.awaitTermination(1, TimeUnit.HOURS);
}
}
}
}

```

Awaiting phase advance#

The `Phaser` class provides a thread/task an opportunity to await the current phase a `Phaser` instance is in via the method `awaitAdvance(int)`. The method takes in the phase argument which a thread intends to wait out. However, the invocation returns immediately if the phase the `Phaser` instance is currently in, is different from the passed-in argument to the method. Consider the program below, where the main thread attempts to await the the very first phase numbered 0. We spawn another thread which sleeps for 5 seconds to simulate work and then records arrival at the barrier. Once it does so, the main thread proceeds forward.

```

import java.util.concurrent.*;

class Demonstration {

    public static void main( String args[] ) throws Exception {
        // create an executor service
        ExecutorService executorService = Executors.newFixedThreadPool(5);

        // create an instance of Phaser class

```

```

Phaser phaser = new Phaser(1);

try {
    executorService.submit(new Runnable() {
        @Override
        public void run() {

            try {
                // sleep for 5 seconds so that the main thread can get a chance to invoke
                // awaitAdvance()
                Thread.sleep(5000);
            } catch (InterruptedException ie) {
                // ignore
            }

            int phase = phaser.arriveAndAwaitAdvance();
            System.out.println(Thread.currentThread().getName() + " arrived at the barrier and
next phase is " + phase);
        }
    });
}

// main thread records arrival at the barrier
int phase;
System.out.println("main thread about to block on phase " + phaser.getPhase());
phase = phaser.awaitAdvance(0);
System.out.println("main thread past the barrier and next phase is " + phase);

} finally {
    executorService.shutdown();
    executorService.awaitTermination(1, TimeUnit.HOURS);
}
}
}
}

```

In the program above, if you change the phase to a higher number e.g. 5 on line#33, the main thread would immediately return from the `awaitAdvance(int)` invocation and the print statement on line#34 will output phase number as 0 instead of 1.

Another example demonstrating the use of `awaitAdvance(int)` appears below. Two parties register with the phaser. The main thread spawns a thread and then `arriveAndDeregister()`-s at the barrier. The call `arriveAndDeregister()` is a non-blocking call and the main thread proceeds forward. Say if we want the main thread to wait for the other worker thread to complete its work and then proceed we can do so by invoking `awaitAdvance(0)` to block the main thread to wait for phase 0 to advance.

```

import java.util.concurrent.*;

class Demonstration {

    public static void main( String args[] ) throws Exception {
        // create an executor service
        ExecutorService executorService = Executors.newFixedThreadPool(5);

        // create an instance of the Phaser class
        Phaser phaser = new Phaser(2);

        try {
            executorService.submit(new Runnable() {
                @Override
                public void run() {

                    try {
                        // sleep for 5 seconds to simulate work
                        Thread.sleep(5000);
                    } catch (InterruptedException ie) {
                        // ignore
                    }

                    int phase = phaser.arriveAndAwaitAdvance();
                    System.out.println(Thread.currentThread().getName() + " arrived at the barrier and
next phase is " + phase);
                }
            });
        }

        // main thread records arrival at the barrier
        int phase = phaser.arriveAndDeregister();
        System.out.println("main thread arrived and deregistered and phase is " + phase);

        // wait for the worker thread to complete work
        phase = phaser.awaitAdvance(0);
        System.out.println("main thread past the barrier and next phase is " + phase);

    } finally {
        executorService.shutdown();
        executorService.awaitTermination(1, TimeUnit.HOURS);
    }
}
}

```

Termination#

A **Phaser** instance can eventually transition to a terminated state. Once terminated, attempts to register with the instance have no effect and all

synchronization methods return immediately with a negative value. One of the ways a `Phaser` can reach a terminal state is when the number of registered parties falls to zero. We can also determine if a `Phaser` is in terminal state by invoking the `isTerminated()` method. The program below demonstrates a `Phaser` instance reaching a terminal state.

```
import java.util.concurrent.*;  
  
class Demonstration {  
    public static void main( String args[] ) {  
        // initialize a `Phaser` with a single party  
        Phaser phaser = new Phaser(1);  
  
        // arrive and deregister so that the registered parties count falls to zero  
        phaser.arriveAndDeregister();  
  
        // print the termination status  
        System.out.println(phaser.isTerminated());  
    }  
}
```

Bear in mind that initializing a `Phaser` with zero registered parties doesn't cause the instance to be in terminated state. The registered party count has to fall from a non-zero count to zero for a `Phaser` to enter into the terminal state. The program below demonstrates these nuances.

```
import java.util.concurrent.*;  
  
class Demonstration {  
  
    public static void main( String args[] ) {  
        // initialize a `Phaser` with zero registered parties  
        Phaser phaser = new Phaser(0);  
  
        // print the termination status  
        System.out.println(phaser.isTerminated());  
  
        // main thread registers with the phaser  
        phaser.register();  
  
        // arrive and deregister so that the registered parties count falls to zero  
        phaser.arriveAndDeregister();  
  
        // print the termination status  
        System.out.println(phaser.isTerminated());  
    }  
}
```

We can also force terminate a `Phaser` instance using the method `forceTermination()`. If so, the waiting threads are abruptly terminated and allowed to proceed past the barrier.

```
import java.util.concurrent.*;  
  
class Demonstration {  
    public static void main( String args[] ) throws Exception {  
        // create an executor service  
        ExecutorService executorService = Executors.newFixedThreadPool(15);  
  
        // register four parties with phaser, when in fact we'll only have  
        // threads synchronize at the barrier to cause them to block  
        Phaser phaser = new Phaser(4);  
        System.out.println("Is phaser terminated " + phaser.isTerminated());  
  
        try {  
            for (int i = 0; i < 3; i++) {  
                executorService.submit(new Runnable() {  
                    @Override  
                    public void run() {  
  
                        phaser.arriveAndAwaitAdvance();  
                        System.out.println(Thread.currentThread().getName() + " proceeding forward.");  
                    }  
                });  
            }  
  
            // wait for worker threads to reach the barrier. We use Thread.sleep only for  
            // demonstration  
            // purposes and keep the examples simple. NEVER use Thread.sleep() to synchronize  
            // between threads!  
            Thread.sleep(3000);  
            System.out.println("Main thread force terminating the phaser.");  
            phaser.forceTermination();  
        } finally {  
            // remember to shutdown the executor  
            executorService.shutdown();  
            executorService.awaitTermination(1, TimeUnit.HOURS);  
        }  
  
        System.out.println("Is phaser terminated " + phaser.isTerminated());  
    }  
}
```

onAdvance()

When a `Phaser` enters the terminal state is determined by a method `onAdvance()` that can also be overridden in derived classes. By default this method returns true when deregistrations cause the number of registered parties to fall to zero. In case, if we never want a `Phaser` to enter the terminal state, we can override the `onAdvance()` method to always return false.

Furthermore, the `onAdvance()` method can be overridden by subclasses to optionally perform some action when the `Phaser` proceeds to the next phase. The `onAdvance()` is executed by one of the threads/parties triggering the phase advance, i.e. the action desired on advancing to the next phase is only performed by one of the threads. Overriding this method is similar to, but more flexible than, providing a barrier action to a `CyclicBarrier`.

To demonstrate the utility of `onAdvance()` method, we'll create a class `MyPhaser` that extends `Phaser` but terminates after 5 iterations or phases, i.e. threads or registered parties are expected to synchronize at the barrier only five times. We'll also print a log statement in the `onAdvance()` method. The code listing appears below with comments:

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

class Demonstration {

    static class MyPhaser extends Phaser {

        public MyPhaser(int registeredParties) {
            super(registeredParties);
        }

        @Override
        protected boolean onAdvance(int phase, int registeredParties) {
            // print the current phase BEFORE advancing and the registered parties
            System.out.println("\n" + Thread.currentThread().getName() + " is performing onAdvance
action. Advancing from phase " + phase + " with registeredParties " + registeredParties);

            // We'll return true after the advance from 4th phase
            // to 5th phase, which will complete the 5 iterations.
            return phase >= 4 || registeredParties == 0;
        }
    }

    public static void main( String args[] ) throws Exception {
        // create an executor service
        ExecutorService executorService = Executors.newFixedThreadPool(15);
```

```

// create an instance of Phaser with 3 registered parties
Phaser phaser = new MyPhaser(3);

try {
    // submit three tasks that'll synchronize on our instance of `MyPhaser`
    for (int i = 0; i < 3; i++) {
        executorService.submit(new Runnable() {
            @Override
            public void run() {
                // each task synchronizes 5 times on the barrier
                for (int i = 0; i < 5; i++) {
                    int phase = phaser.arriveAndAwaitAdvance();
                    System.out.println(Thread.currentThread().getName() + " has advanced to
phase " + phase);
                }
            }
        });
    }

} finally {
    // remember to shutdown the executor
    executorService.shutdown();
    executorService.awaitTermination(1, TimeUnit.HOURS);
}

System.out.println("is terminated " + phaser.isTerminated());
}
}

```

The output of the above program shows that one of the threads executes the `onAdvance()` method and threads/tasks advance from one phase to another when repeatedly synchronizing on the barrier. The interesting aspect to note is that when we ask for the terminal status at the end of the program, it is reported as true and two of the threads print negative phase numbers as return values from the `arriveAndAwaitAdvance()` method. The return values indicate the next phase number and a negative value implies that the phaser is now in terminal state. The sixth attempt to synchronize on the barrier will fail. You can change the limit to 6 on line#39 in the `for` loop to observe the behavior of the barrier in terminal state. The invocations to `arriveAndAwaitAdvance()` methods return immediately with a negative value.

The above program can be rewritten without a separate class as shown in the widget below. Each thread/task checks for the phaser's terminal condition in a while loop and performs some action repeatedly. This is the

idiomatic use of overriding the `onAdvance()` method and its default functionality.

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

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

        // create an executor service
        ExecutorService executorService = Executors.newFixedThreadPool(15);

        // create an instance of Phaser with 3 registered parties
        Phaser phaser = new Phaser() {
            protected boolean onAdvance(int phase, int registeredParties) {
                // print the current phase BEFORE advancing and the registered parties
                System.out.println("\n" + Thread.currentThread().getName() + " is performing
onAdvance action. Advancing from phase " + phase + " with registeredParties " +
registeredParties);

                // return true when 5 iterations are complete
                return phase >= 4 || registeredParties == 0;
            }
        };

        try {

            // register the threads that'll synchronize on the barrier
            phaser.bulkRegister(3);

            // submit three tasks that'll synchronize on our instance of `MyPhaser`
            for (int i = 0; i < 3; i++) {
                executorService.submit(new Runnable() {
                    @Override
                    public void run() {

                        // repeatedly synchronize until the barrier is in terminated state
                        while (!phaser.isTerminated()) {
                            int phase = phaser.arriveAndAwaitAdvance();
                            System.out.println(Thread.currentThread().getName() + " has advanced to
phase " + phase);
                        }
                    }
                });
            }
        } finally {
            // remember to shutdown the executor
            executorService.shutdown();
            executorService.awaitTermination(1, TimeUnit.HOURS);
        }
    }
}
```

```

    }

    System.out.println("is terminated " + phaser.isTerminated());
}
}

```

Tiering#

The **Phaser** allows registration of a maximum number of 65,535 parties. For scenarios, where we want to register a greater number of parties than allowed, we can construct *tiered phasers* to accommodate an arbitrarily large set of participants. Tiering refers to arranging **Phaser** instances as a tree and establishing child-parent relationships between them.

Apart from circumventing the limitation on the number of possible registered parties, tiered phasers can reduce the heavy synchronization contention experienced by **Phaser**-s with large number of parties. Participants can be divided among group of sub-phasers that share a common parent. Doing so may greatly increase throughput even though it incurs greater per-operation overhead.

In a tree of tiered phasers, registration and deregistration of child phasers with their parent phaser are managed automatically. Whenever the number of registered parties of a child phaser becomes non-zero the child phaser is registered with its parent and vice versa, i.e. the child phaser is deregistered with its parent in case the registered parties for the child become zero.

The widget below illustrates a simple program showing the working of parent and child phasers:

```

import java.util.concurrent.*;

class Demonstration {

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

        // create an executor service
        ExecutorService executorService = Executors.newFixedThreadPool(15);

        // initially create a parent phaser which has no registered party
        Phaser phaserParent = new Phaser(0);

        // child phaser 1
        Phaser childPhaser1 = new Phaser(phaserParent, 1);

        // child phaser 2
        Phaser childPhaser2 = new Phaser(phaserParent, 2);
    }
}

```

```

// child phaser 3
Phaser childPhaser3 = new Phaser(phaserParent, 3);

// register the main thread too
phaserParent.register();

System.out.println("Registered party count for parentPhaser " +
phaserParent.getRegisteredParties());

try {
    // arrive at child phaser 1
    executorService.submit(new Runnable() {
        @Override
        public void run() {
            childPhaser1.arriveAndAwaitAdvance();
            System.out.println("Thread 1 moving ahead");
        }
    });
}

// arrive at child phaser 2
executorService.submit(new Runnable() {
    @Override
    public void run() {
        childPhaser2.arrive();
        childPhaser2.arriveAndAwaitAdvance();
        System.out.println("Thread 2 moving ahead");
    }
});

// arrive at child phaser 3
executorService.submit(new Runnable() {
    @Override
    public void run() {
        childPhaser3.arrive();
        childPhaser3.arrive();
        childPhaser3.arriveAndAwaitAdvance();
        System.out.println("Thread 3 moving ahead");
    }
});

// main thread arrives at the parent phaser
phaserParent.arriveAndAwaitAdvance();

System.out.println("main thread existing.");

} finally {

```

```

        // remember to shutdown the executor
        executorService.shutdown();
        executorService.awaitTermination(1, TimeUnit.HOURS);
    }
}

}

```

If you run the above program, you'll see that the number of registered parties for the `parentPhaser` is output to be 4. We create a parent-child relationship between the `parentPhaser` and the other three phasers by passing in the `parentPhaser` as the argument for the parent in the `Phase` constructor. It is important to note that from the parent phaser's perspective the number of registered parties is 4 which includes the main thread and the other three phasers. *The registered parties for each individual child phaser don't factor into the count of registered parties for the parent phaser.* In fact, the parent phaser moves to the next phase when all the child phases move to the next phase and the main thread records arrival with the parent phaser. However, this also implies that any thread/task that invokes `arriveAndAwaitAdvance()` on a child phase will not move ahead until the root phase moves to the next phase. Awaiting advance on any child phaser amounts to awaiting advance on all the children of the root phaser.

As an exercise if you comment out lines 33 and 34 in the above widget and re-run the program, you'll observe the program will hang and execution will time out. The other thread, even though they arrive at their respective child phasers but don't move past the child barrier since the root phaser is still waiting for an unarrived party.

Another caveat when working with tiered phasers is to be cognizant that if the parent phaser advances its phase then all the child phasers do too, even if the required number of parties have not arrived at the child phaser. This is illustrated by the following program, where threads block on child phasers proceed past the barrier when the main thread artificially causes a phase advance for the parent phaser.

```

import java.util.concurrent.*;

class Demonstration {

    public static void main( String args[] ) throws Exception {
        // create an executor service
        ExecutorService executorService = Executors.newFixedThreadPool(15);

        // initially create a parent phaser which has no registered party
        Phaser phaserParent = new Phaser(0);

```

```

// child phaser 1
Phaser childPhaser1 = new Phaser(phaserParent, 1);

// child phaser 2
Phaser childPhaser2 = new Phaser(phaserParent, 2);

// child phaser 3
Phaser childPhaser3 = new Phaser(phaserParent, 3);

// register the main thread too
phaserParent.register();

System.out.println("Registered party count for parentPhaser " +
phaserParent.getRegisteredParties());

try {
    // arrive at child phaser 1
    executorService.submit(new Runnable() {
        @Override
        public void run() {
            childPhaser1.arriveAndAwaitAdvance();
            System.out.println("Thread 1 moving ahead");
        }
    });
}

// arrive at child phaser 2
executorService.submit(new Runnable() {
    @Override
    public void run() {
        childPhaser2.arriveAndAwaitAdvance();
        System.out.println("Thread 2 moving ahead");
    }
});

// arrive at child phaser 3
executorService.submit(new Runnable() {
    @Override
    public void run() {
        childPhaser3.arriveAndAwaitAdvance();
        System.out.println("Thread 3 moving ahead");
    }
});

// wait for threads to block on child phasers
Thread.sleep(3000);
phaserParent.arrive();
phaserParent.arrive();

```

```

phaserParent.arrive();
phaserParent.arrive();

System.out.println("main thread existing.");

} finally {
    // remember to shutdown the executor
    executorService.shutdown();
    executorService.awaitTermination(1, TimeUnit.HOURS);
}
}
}
}

```

An involved example#

Let's consider a hypothetical example of producer and consumer threads. Suppose our program flow spawns a bunch of producer and consumer threads and we want to structure the program such that the producer threads run first, the consumer threads second and finally the main thread exits after the two sets of threads/tasks are done. The program along with detailed comments appears in the widget below:

```

import java.util.concurrent.*;

class Demonstration {

    public static void main( String args[] ) throws Exception {
        // create an executor service
        ExecutorService executorService = Executors.newFixedThreadPool(15);

        // initially create a parent phaser which has no registered party
        Phaser phaserParent = new Phaser(0);
        // producer phaser has three parties, one for each producer
        Phaser producersPhaser = new Phaser(phaserParent, 3);

        // consumer phaser has five parties, including the three producers
        // so that we can have the consumers wait until the producers are done
        Phaser consumerPhaser = new Phaser(phaserParent, 2);

        phaserParent.register();

        System.out.println("Registered party count for parentPhaser " +
phaserParent.getRegisteredParties());

        try {
            // create 3 producer threads

```

```

for (int i = 0; i < 3; i++) {
    executorService.submit(new Runnable() {
        @Override
        public void run() {

            // all producers reach barrier and then start producing
            producersPhaser.arriveAndAwaitAdvance();

            // ... work to produce.

            System.out.println(Thread.currentThread().getName() + " producer finished at
parent phase " + phaserParent.getPhase());

            // Now wait for consumers to get done.
            producersPhaser.arrive();
            phaserParent.awaitAdvance(1);

            // unblock the main thread
            producersPhaser.arrive();

        }
    });
}

// create two consumer threads
for (int i = 0; i < 2; i++) {
    executorService.submit(new Runnable() {
        @Override
        public void run() {

            // wait for producers to get done
            while (phaserParent.getPhase() <= 1) {
                consumerPhaser.arriveAndAwaitAdvance();
            }

            // ... work to consume

            System.out.println(Thread.currentThread().getName() + " consumer finished at
parent phase " + phaserParent.getPhase());

            // Now unblock the main thread
            consumerPhaser.arrive();

        }
    });
}

// get the producers going

```

```

phaserParent.arrive();

// wait for the producers to be done
phaserParent.awaitAdvance(0);

// get the consumers going
phaserParent.arrive();

// wait for consumers to get done
phaserParent.awaitAdvance(1);

// wait for both consumer and producers to exit
phaserParent.arriveAndAwaitAdvance();

System.out.println("main thread existing at parent phase " + phaserParent.getPhase());

} finally {
    // remember to shutdown the executor
    executorService.shutdown();
    executorService.awaitTermination(1, TimeUnit.HOURS);
}
}
}
}

```

Conclusion#

In summary, [Phaser](#) is an advanced and sophisticated barrier construct that can be used in complex synchronization scenarios, however, beware its use can also introduce subtle bugs that may be hard to find. For simple use cases go with the simpler [CyclicBarrier](#) or [CountDownLatch](#).

IllegalMonitorStateException

Learn the reasons that cause `IllegalMonitorStateException` to be thrown.

Explanation#

The [IllegalMonitorStateException](#) is a common programming error that can show up in concurrent programs. Depending on the structure of the program, the exception may occur consistently or only occasionally. The [IllegalMonitorStateException](#) exception class extends

the `RuntimeException` class and according to the official documentation is thrown to indicate that a thread has attempted to wait on an object's monitor or to notify other threads waiting on an object's monitor without owning the specified monitor. In other words if you invoke `wait()` or `notify()/notifyAll()` without synchronizing on the object i.e. outside of a `synchronized` method or block (with the object as the synchronization target) then `IllegalMonitorStateException` will be thrown. Similarly, the exception is thrown when you invoke these methods on an instance of `Condition` class without acquiring the associated lock with the condition. The class is part of the `java.lang` package and not `java.util.concurrent.*`.

Repro using `Lock`#

The program below demonstrates generating `IllegalMonitorStateException` using a `Condition` object that was instantiated from a `Lock` object.

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

class Demonstration {
    public static void main( String args[] ) throws Exception {
        Lock lock = new ReentrantLock();
        Condition condition = lock.newCondition();

        // throws exception because we didn't lock the associated
        // lock object with the condition variable before invoking
        // await() on the condition object.
        condition.await();
    }
}
```

The fix for the above program appears below:

```
// acquire the associated lock
lock.lock();
try {
    while (/* some condition */ ) {
        // always invoke await or wait in a loop to cater for spur
ious wake-ups
        // and after synchronizing on the associated loc
k
        condition.await();
    }
} finally {
    // remember to unlock in a finally block
}
```

```
        lock.unlock();
    }
```

Repro using object#

The program in the widget below causes `IllegalMonitorStateException` by invoking `notifyAll()` on an object without synchronizing on it.

```
class Demonstration {
    public static void main( String args[] ) throws Exception {
        Object myObject = new Object();

        // throws exception because we didn't synchronize
        // on myObject before invoking the wait() method
        myObject.notifyAll();
    }
}
```

The fix for the above program appears below:

```
Object myObject = new Object();

synchronized (myObject) {
    // invoking notifyAll() in a block synchronized on myObject
    myObject.notifyAll();
}
```

Consider the program below and state the outcome of executing the `main()` method.

```
public class IllegalMonitorStateQuiz {

    synchronized void someFunction() throws InterruptedException {
        this.wait();
    }

    public static void main(String[] args) throws Exception {
        (new IllegalMonitorStateQuiz()).someFunction();
    }
}
```

A)

`IllegalMonitorStateException` is thrown.

B)

`InterruptedException` is thrown.

C)

Program is either blocked on `wait()` forever or exits because of a spurious wakeup.

D)

The target of `synchronized` and the object invoking `wait()` are different and the program doesn't compile.

Consider the program below:

```
class Program {  
    synchronized void someFunction() {  
        Object object = new Object();  
        synchronized (this) {  
            object.notify();  
        }  
    }  
}
```

What is the outcome of running `someFunction()`?

A)

`InterruptedException` is thrown.

B)

Program doesn't compile.

C)

Program ends in a deadlock.

D)

`IllegalMonitorStateException` is thrown

TimeoutException

Learn the reasons that cause `TimeoutException` to be thrown.

`TimeoutException` is used as a means to indicate that a blocking operation has timed-out. Consider the `CyclicBarrier` class (*read about cyclic barrier [here](#)*) which exposes the method `await()`, which takes in a timeout value. If a thread invokes this method on an object of `CyclicBarrier` and the barrier isn't reached by other threads within the specified timeout by the first thread then `TimeoutException` is thrown and the barrier is broken. In

some cases, values such as boolean can be returned to indicate timeout rather than throwing the `TimeoutException`.

The program in the widget below demonstrates an instance of a `CyclicBarrier` throwing the `TimeoutException` when the main thread awaits at the barrier for 100 milliseconds.

```
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;
import java.util.concurrent.BrokenBarrierException;

class Demonstration {
    public static void main( String args[] ) throws TimeoutException, InterruptedException,
BrokenBarrierException {

        // create a barrier for two threads.
        CyclicBarrier barrier = new CyclicBarrier(2);

        // TimeoutException thrown after 100 ms elapse
        barrier.await(100, TimeUnit.MILLISECONDS);

    }
}
```

`TimeoutException` is a checked exception extending the `Exception` class and is part of the `java.util.concurrent` package. Some other Java classes that throw the `TimeoutException` are:

- `java.util.concurrent.AbstractExecutorService`
- `jdk.jshell.execution.JdiInitiator`
- `sun.nio.ch.PendingFuture`

CancellationException

Learn the reasons that cause `CancellationException` to be thrown.

Overview#

`CancellationException` is thrown to indicate that the output or result of a value producing task can't be retrieved because it was cancelled. `CancellationException` is an unchecked exception and extends `IllegalStateException`, which in turn extends the `RuntimeException`. Some of the classes that throw `CancellationException` exception are:

- FutureTask
- CompletableFuture
- ForkJoinTask

Example#

The following program demonstrates a scenario where an instance of `CancellationException` is thrown by the program. We create a `FutureTask` that has the thread sleep for one second intervals for a total of one hour. The main thread submits the task to the executor service, waits for three seconds and then attempts to cancel the task. Next when the main thread attempts to retrieve the result of the task by invoking the `get()` method on the task object, `CancellationException` exception is thrown.

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

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

        ExecutorService es = Executors.newFixedThreadPool(5);

        // Create a FutureTask, which takes in an instance of Callable
        FutureTask<Integer> futureTask = new FutureTask<>(new Callable<Integer>() {
            @Override
            public Integer call() throws Exception {

                // The task simply intends to sleep for an hour but one second at a time
                for (int i = 0; i < 3600; i++) {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException ie) {
                        System.out.println("Thead " + Thread.currentThread().getName() + " has been
interrupted");
                        break;
                    }
                }
                return 786;
            }
        });

        try {
            es.submit(futureTask);
            Thread.sleep(3000);
            futureTask.cancel(true);
            futureTask.get();
        }
```

```

        } catch (InterruptedException | ExecutionException e) {
            // ignore exceptions since this is a demo
        } finally {
            // remember to shutdown the executor service
            es.shutdown();
        }
    }
}

```

Some other nuances about the above program include:

- If the main thread doesn't invoke `task.get()` then `CancellationException` isn't thrown even if the main thread invokes `cancel()` on the task object.
- The `cancel()` method on the task object takes in a boolean `mayInterruptIfRunning`, indicating if the task should be interrupted in case it is running. In the above program if we pass `false` instead of `true` on line#29, the main thread will still see the `CancellationException` being thrown by the `get()` method, however, the task keeps running and the print statement on line#18 doesn't appear in the program output

ExecutionException

Guide to working with ExecutionException.

Overview#

`ExecutionException` is a checked exception that extends the `Exception` class. This exception is thrown by an instance of `FutureTask` that encounters an `Exception` or `Error` (both derivates of `Throwable`) that remain unhandled by user code and, subsequently an attempt is made to retrieve the result of such a task.

Example#

Consider the program below that has a task submitted to the `ExecutorService`. The task simply throws a runtime exception to simulate failure. When we attempt to retrieve the result of the task, the snippet `futureTask.get()` throws `ExecutionException`.

```

import java.util.concurrent.*;

class Demonstration {

```

```

public static void main( String args[] ) {
    ExecutorService es = Executors.newFixedThreadPool(5);

    // Create a FutureTask, which takes in an instance of Callable
    FutureTask<Integer> futureTask = new FutureTask<>(new Callable<Integer>() {
        @Override
        public Integer call() throws Exception {
            // The task simulates encountering an exception by throwing one.
            throw new RuntimeException("runtime issue");
        }
    });

    try {
        es.submit(futureTask);

        // wait a while for the task
        Thread.sleep(300);

        // Attempt to get the result of the task
        futureTask.get();
    } catch (ExecutionException e) {
        // You should see the following line print in the console.
        System.out.println("ExecutionException thrown by program.");
    } catch (InterruptedException ie) {
        // we can ignore InterruptedException for demo purposes
    } finally {
        // remember to shutdown the executor service
        es.shutdown();
    }
}
}

```

In the above example, if you comment out **line#23** and the associated catch clause for **ExecutionException**, upon re-run of the program you'll notice that it doesn't throw **ExecutionException** even though the task throws a runtime exception. A programming oversight to retrieve the result of submitted task that fail can falsely lead the program to exit successfully.

Finally, if we replace **line#12** with the snippet `throw new Error();`, we'll still observe the **ExecutionException** being thrown upon retrieving the result of the program.

RejectedExecutionException

Learn the causes of RejectedExecutionException being thrown.

Overview#

Rather than creating individual threads for executing jobs in parallel, we can leverage Java's executor classes that abstract away thread scheduling and management from the user and leave the user to focus on submitting `Runnable` or `Callable` tasks/commands. Some of the classes implementing the `Executor` and `ExecutorService` interfaces are as follows:

- `AbstractExecutorService`
- `ForkJoinPool`
- `ScheduledThreadPoolExecutor`
- `ThreadPoolExecutor`

The `Executor` interface defines a method `execute` while the `ExecutorService` defines overloaded versions of the `submit` method. Both these methods throw the runtime exception `RejectedExecutionException` when implemented by the various executor classes. The `RejectedExecutionException` exception is thrown when a task can't be accepted by the executor for execution. Generally there are two scenarios when a task is rejected for execution: The executor service has already been shut down. When the executor service has exhausted resources and can't take on any more task submissions. For instance in the case of the `ThreadPoolExecutor`, the `RejectedExecutionException` is thrown when the executor service uses a queue with a defined maximum capacity and a defined maximum number of threads for the thread pool and both resources have reached capacity.

Example#

Consider the program below that attempts to submit a task for execution to a fixed-size thread pool executor, after `shutdown()` has already been invoked. The executor throws the `RejectedExecutionException`.

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

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

        // create a executor service
        ExecutorService executor = Executors.newFixedThreadPool(5);

        // shutdown the executor
        executor.shutdown();

        // attempt to execute a Runnable after the ExecutorService has been shutdown
        executor.execute(new Runnable() {
```

```

    @Override
    public void run() {
        // dummy task
    }
});
}
}

```

As a second example, consider the program below that uses an instance of `ThreadPoolExecutor` as the executor service to submit tasks for execution. We instantiate the instance with a thread pool size of 10 and a queue to hold submitted tasks of size 10 too. The first 10 tasks submitted are all executed by the 10 available threads and the queue remains empty. However, the next ten tasks submitted fill-up the queue and on submitting the 21st task the executor throws `RejectedExecutionException`.

```

import java.util.concurrent.*;

class Demonstration {

    public static void main( String args[] ) {

        // Create a ThreadPoolExecutor with maximum of 10 threads and a queue that can hold at
        // most 10 tasks.
        ExecutorService executorService = new ThreadPoolExecutor(5, 10, 1, TimeUnit.HOURS,
            new LinkedBlockingQueue<Runnable>(10));

        // declare outside the try loop to determine which task gets rejected.
        int i = 0;
        try {
            // The first ten tasks should be executed by the ten threads in the pool. The next ten
            // tasks should
            // fill up the queue while the 21st task should cause the executor service to throw the
            RejectedExecutionException.

            for (; i < 21; i++) {
                executorService.execute(new Runnable() {
                    @Override
                    public void run() {
                        try {
                            // Each task simulates work by sleeping for a minute.
                            Thread.sleep(1000 * 60);
                        } catch (InterruptedException ie) {
                            // This statement will print on the console when we shutdown the executor
                            before the tasks complete.
                            System.out.println("Task executed by thread " +
                                Thread.currentThread().getName() + " has been interrupted");
                        }
                    }
                });
            }
        }
    }
}

```

```

        }
    });
}
// This statement never gets printed.
System.out.println("Submitted all tasks");
} catch (RejectedExecutionException ree) {
    System.out.println("Task " + i + " was rejected");
} finally {
    // force the executor to stop all executing tasks and shutdown.
    executorService.shutdownNow();
}

}
}

```

In the above program if you change the for loop limit from 21 to 20 on line#16, the program would run without experiencing any exceptions.

CompletionException

Guide to understanding CompletionException

Overview#

The `CompletionException` extends from the `RuntimeException` and is thrown when a task hits an error or exception when executing.

The `CompletionException` and the `ExecutionException` might seem similar but there's a fundamental difference between the two. The `ExecutionException` is thrown only when an attempt is made to retrieve the result/outcome of a task, while the `CompletionException` can be thrown when waiting on a task to complete or when trying to retrieve a result computed by a task. In a sense `CompletionException` can be thought of as broader in scope than `ExecutionException`.

Example#

We'll use the `CompletableFuture` class to demonstrate the `CompletionException`. The `CompletableFuture` class is used to program asynchronously in Java. It exposes a method `completeExceptionally` that takes in an instance of `Throwable` to indicate the failure experienced by the task. When the main thread attempts to wait for the task to complete by invoking the `join()` method the `CompletionException` is thrown.

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

```

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

        // create an executor service and a completable future
        ExecutorService executorService = Executors.newFixedThreadPool(5);
        CompletableFuture<Integer> completableFuture = new CompletableFuture<>();

        try {
            // create a command that marks the CompletableFuture instance complete exceptionally
            Runnable command = new Runnable() {
                @Override
                public void run() {
                    completableFuture.completeExceptionally(new Exception("Something bad
happened."));
                }
            };

            // submit the command
            executorService.submit(command);

            // wait for the future to complete. This statement throws the CompletionException
            Thread.sleep(500);
            completableFuture.join();
        } catch (CompletionException ce) {
            System.out.println("CompletionException has been thrown." + ce.getCause());
        } catch (InterruptedException ie) {
            // ignore for now
        } finally {
            // remember to shutdown the executor service in a finally block
            executorService.shutdown();
        }
    }
}

```

Note that in the above program, if we attempt to retrieve the result of the task, which should be an integer we'll get **ExecutionException** instead of the **CompletionException**.

```

import java.util.concurrent.*;

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

        // create an executor service and a completable future
        ExecutorService executorService = Executors.newFixedThreadPool(5);
        CompletableFuture<Integer> completableFuture = new CompletableFuture<>();

        try {
            // create a command that marks the CompletableFuture instance complete exceptionally
            Runnable command = new Runnable() {

```

```

@Override
public void run() {
    completableFuture.completeExceptionally(new Exception("Something bad
happened."));
}
};

// submit the command
executorService.submit(command);

// wait for the future to complete. This statement throws the CompletionException
Thread.sleep(500);
completableFuture.get();

} catch (CompletionException ce) {
    System.out.println("CompletionException has been thrown." + ce.getCause());
} catch (ExecutionException ee) {
    System.out.println("Execution Exception has been thrown " + ee.getCause());
} catch (InterruptedException ie) {
    // ignore for now
} finally {
    // remember to shutdown the executor service in a finally block
    executorService.shutdown();
}
}
}
}

```

The `CompletableFuture` also offers another method `getNow()` that is non-blocking unlike the `get()` method. The `getNow()` method either returns the computed value if the future has completed or a default value that is passed-in at the time of the invocation of the method. The `getNow()` method throws the `CompletionException` instead of the `ExecutionException` in case the future is exceptionally completed as shown in the following program.

```

import java.util.concurrent.*;

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

        // create an executor service and a completable future
        ExecutorService executorService = Executors.newFixedThreadPool(5);
        CompletableFuture<Integer> completableFuture = new CompletableFuture<>();

        try {
            // create a command that marks the CompletableFuture instance complete exceptionally
            Runnable command = new Runnable() {
                @Override
                public void run() {
                    completableFuture.completeExceptionally(new Exception("Something bad
happened."));
                }
            }
        }
    }
}

```

```
};

// submit the command
executorService.submit(command);

// wait for the future to complete. This statement throws the CompletionException
Thread.sleep(500);
completableFuture.getNow(-1);

} catch (CompletionException ce) {
    System.out.println("CompletionException has been thrown." + ce.getCause());
} catch (InterruptedException ie) {
    // ignore for now
} finally {
    // remember to shutdown the executor service in a finally block
    executorService.shutdown();
}

}
```

Interestingly, if the `CompletableFuture` completes successfully but then throws an exception, the `join()` and `get()` methods don't throw any exception in the main thread as demonstrated by the following program.

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

```
class Demonstration {  
    public static void main( String args[] ) {  
  
        // create an executor service and a completable future  
        ExecutorService executorService = Executors.newFixedThreadPool(5);  
        CompletableFuture<Integer> completableFuture = new CompletableFuture<>();  
  
        try {  
            // create a command that marks the CompletableFuture instance complete exceptionally  
            Runnable command = new Runnable() {  
                @Override  
                public void run() {  
                    // complete the future successfully  
                    completableFuture.complete(786);  
  
                    // throw an exception now  
                    throw new RuntimeException("Something bad happened.");  
  
                }  
            };  
  
            // submit the command  
            executorService.submit(command);  
  
            // wait for the future to complete. This statement throws the CompletionException  
        } catch (CompletionException e) {  
            System.out.println("The future completed exceptionally: " + e.getMessage());  
        }  
    }  
}
```

```

        Thread.sleep(500);
        completableFuture.get(); // or completableFuture.join()
    } catch (ExecutionException ce) {
        System.out.println("ExecutionException has been thrown." + ce.getCause());
    } catch (InterruptedException ie) {
        // ignore for now
    } finally {
        // remember to shutdown the executor service in a finally block
        executorService.shutdown();
    }
}
}
}

```

BrokenBarrierException

Understand the causes of BrokenBarrierException with runnable code examples in the browser.

Overview#

The `BrokenBarrierException` is usually an indication of a programming flaw. It occurs when:

1. A thread is already waiting on a barrier and the barrier enters the broken state.
2. A thread attempts to wait on a barrier that is in the broken state.

Example#

Consider the program below that submits two tasks to the executor. Both the tasks `await()` on a `CyclicBarrier` object, which is initialized with a count of 3, i.e. three threads must `await()` the barrier object before they can proceed forward. The main thread `cancel()`-s one of the tasks. The thread executing the task is already waiting on the barrier and experiences an `InterruptedException`. At this point the barrier is broken and the second task throws `BrokenBarrierException`. This sequence of events manifests the first condition described above i.e. `BrokenBarrierException` is thrown when a barrier is broken and there are threads already waiting on the barrier object.

The other condition when a `BrokenBarrierException` is thrown, is when a thread attempts to `await()` an already broken barrier. This is showcased by

the main thread when it `await()`-s the broken barrier object on **line#55** and the program exits with the `BrokenBarrierException`.

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

class Demonstration {

    public static void main( String args[] ) throws InterruptedException, BrokenBarrierException {

        CyclicBarrier barrier = new CyclicBarrier(3);
        ExecutorService executorService = Executors.newFixedThreadPool(5);

        Runnable task1 = new Runnable() {
            @Override
            public void run() {
                try {
                    barrier.await();
                } catch (BrokenBarrierException bbe) {
                    System.out.println("Broken barrier exception experienced by " +
Thread.currentThread().getName());
                } catch (InterruptedException ie) {
                    System.out.println("Interrupted exception experienced by " +
Thread.currentThread().getName());
                }
            }
        };

        Runnable task2 = new Runnable() {
            @Override
            public void run() {
                try {
                    barrier.await();
                } catch (BrokenBarrierException bbe) {
                    System.out.println("Broken barrier exception experienced by " +
Thread.currentThread().getName());
                } catch (InterruptedException ie) {
                    System.out.println("Interrupted exception experienced by " +
Thread.currentThread().getName());
                }
            }
        };

        try {
            Future future1 = executorService.submit(task1);

            executorService.submit(task2);
        }
```

```
// wait for other threads to reach the barrier. Never use Thread.sleep() for synchronization
among
// threads. This example uses it only for demonstration purposes.
Thread.sleep(1000);

// cancel the first task so that the thread executing it is interrupted while waiting on the
barrier
future1.cancel(true);

// wait for both other threads to finish.
Thread.sleep(1000);

// check the state of the barrier
System.out.println("Is barrier broken " + barrier.isBroken());

// attempt to wait on an already broken barrier
barrier.await();
} finally {
    // remember to shut down the executor
    executorService.shutdown();
}

}
```

Annotations

Learn the use of various annotations used in concurrent code.

We'll cover the following



- Overview
- Class-level annotations
 - `@ThreadSafe`
 - `@NotThreadSafe`
 - `@Immutable`
- Field and method-level annotations
 - `@GuardedBy(lock)`
 - `@GuardedBy("this")`
 - `@GuardedBy("field")`
 - `@GuardedBy("ClassName.fieldName")`
 - `@GuardedBy("methodName())`
 - `@GuardedBy("ClassName.class")`
- Summary

Overview#

In this lesson, we'll explore the various Java annotations that are related to concurrency. These are:

1. `@ThreadSafe`
2. `@NotThreadSafe`
3. `@Immutable`
4. `@GuardedBy(lock)`

Class-level annotations#

Note that these annotations serve as documentation about class behavior but don't change the ability or functionality of class in any way. Also, these class-level annotations become part of the public documentation of a class.

@ThreadSafe#

The annotation `@ThreadSafe` can be applied to a class to indicate to users and maintainers that the class is thread safe and multiple threads can concurrently interact with an instance of the class without worrying about synchronization. Newbies should not confuse the annotation to mean that it makes a class thread safe. The annotation only serves as documentation.

@NotThreadSafe#

The `@NotThreadSafe` annotation is the opposite of `@ThreadSafe` and is intended to explicitly communicate to the users and maintainers that the class requires synchronization effort on part of the users to ensure thread-safe concurrent access to instances of the class.

@Immutable#

The `@Immutable` annotation indicates that once an object of the class is instantiated, it can't be changed. All operations on such an instance of the class become read-only operations and consequently the class becomes thread safe. In other words, `@Immutable` implies `@ThreadSafe`. The `String` class is, perhaps, the most well-known Java immutable class. Other examples include wrapper classes such as `Integer`, `Byte`, `Short`, and `Boolean` Java classes.

Field and method-level annotations#

Unlike class-level annotations the field and method-level annotations don't appear in the public documentation of a class and are targeted towards informing the maintainers and extenders of a class.

@GuardedBy(lock)

The annotation `@GuardedBy(lock)` can be used to document that a certain field or method should only be accessed while holding the lock argument in the annotation. The `lock` argument passed-in to the annotation can take-on different values which we'll discuss below:

Consider the class `AnnotationsExampleClass` in the widget below:

```
public class AnnotationsExampleClass {

    // field is protected by the method instance
    @GuardedBy("this") // example of @GuardedBy("this")
    private String[] stringValues;

    private Object customLock = new Object();

    private static ReentrantLock counterLock = new ReentrantLock();

    @GuardedBy("AnnotationsExampleClass.class") // example of
    @GuardedBy("ClassName.class")
    private static long lastAccess = System.currentTimeMillis();

    private void updateAccessTime() {
        synchronized (AnnotationsExampleClass.class) {
            lastAccess = System.currentTimeMillis();
        }
    }

    public void manipulateStringValues() {
        // This method holds the "this" lock before
        // accessing the stringValues variable.
        synchronized (this) {
            // ... modify array of Strings here
        }
    }

    // method is protected by the object instance
    @GuardedBy("customLock") // example of @GuardedBy("field")
    private String getStringValue(int index) {
        return stringValues[index];
    }

    // This method acquires two locks
    synchronized public void printValues() {

        for (int i = 0; i < stringValues.length; i++) {
            synchronized (customLock) {
                getStringValue(i);
            }
        }
    }

    // method used to return an appropriate lock based on the input argument.
    private Object lockMaster(int input) {
```

```

switch (input) {

    case 1:
        return customLock;

        // ... other cases
    }

    return null;
}

{@GuardedBy("lockMaster(1)") // @GuardedBy("methodName()")}

private void transforms() {
    // ... method should be invoked only after acquiring the lock
    // ... returned by the lockMaster method
}

static class Helper {
    {@GuardedBy("AnnotationsExampleClass.counterLock") // example of
@GuardedBy("ClassName.fieldName")
    private static int counter = 0;

    public void incrementCounter() {
        try {
            AnnotationsExampleClass.counterLock.lock();
            counter++;
        } finally {
            AnnotationsExampleClass.counterLock.unlock();
        }
    }
}

```

@GuardedBy("this")#

Indicates that the field or method should be accessed while holding the intrinsic lock, i.e. the object itself of which the field or method is a member. In our example above the field `stringValues` uses this annotation and in the method `manipulateStringValues()` the intrinsic lock is acquired before manipulating `stringValues`.

`@GuardedBy("field")#`

Indicates that the field or method should be accessed while synchronizing on the lock associated with the named field in the annotation. In our example class `AnnotationsExampleClass`, the field `customLock` appears as the named field in the annotation for the method `getStringValue()`. This method is in turn invoked by `printValues()` which acquires the intrinsic lock associated with the `customLock` field before invoking the `getStringValue()` method.

If the named field in the annotation were a lock object itself, e.g. if `customLock` were an instance of the `ReentrantLock` class we'd acquire the explicit lock, i.e. `customLock.lock()` rather than the intrinsic lock associated with the object.

`@GuardedBy("ClassName.fieldName")#`

This form of annotation is similar to the `@GuardedBy("field")`, however, the lock argument represents an object held in a static field of another class. In our example, the inner `Helper` class's `counter` field uses the annotation to indicate to the users to acquire the `counterLock` of the `AnnotationsExampleClass` before manipulating the `counter` object.

`@GuardedBy("methodName()")#`

Indicates that the lock returned by the named method should be acquired before interacting with the annotated field or method. In our example class, the method `lockMaster()` is named in the annotation for the method `transforms()`. The annotation implies that `lockMaster(1)` should be invoked, the returned lock acquired and then only `transforms()` should be invoked.

`@GuardedBy("ClassName.class")#`

Indicates that the annotated field or method should be accessed only after acquiring the class literal object for the named class. In our example, the field `lastAccess` is annotated with the `@GuardedBy` annotation with the class object `AnnotationsExampleClass`. The method `updateAccessTime()` correctly acquires the lock on the class object before accessing the `lastAccess` field.

Summary#

The annotations we discussed in this lesson only serve to document notes and information for users and maintainers but can also be beneficial for static analysis tools that can programmatically verify if a class behaves as

advertised. For instance, tools can verify if a class marked as `@Immutable` is truly immutable or not. In general, it is a good practise to annotate where appropriate for improving readability of one's code.

`@GuardedBy("this")#`

Indicates that the field or method should be accessed while holding the intrinsic lock, i.e. the object itself of which the field or method is a member. In our example above the field `stringValues` uses this annotation and in the method `manipulateStringValues()` the intrinsic lock is acquired before manipulating `stringValues`.

`@GuardedBy("field")#`

Indicates that the field or method should be accessed while synchronizing on the lock associated with the named field in the annotation. In our example class `AnnotationsExampleClass`, the field `customLock` appears as the named field in the annotation for the method `getStringValue()`. This method is in turn invoked by `printValues()` which acquires the intrinsic lock associated with the `customLock` field before invoking the `getStringValue()` method.

If the named field in the annotation were a lock object itself, e.g. if `customLock` were an instance of the `ReentrantLock` class we'd acquire the explicit lock, i.e. `customLock.lock()` rather than the intrinsic lock associated with the object.

`@GuardedBy("ClassName.fieldName")#`

This form of annotation is similar to the `@GuardedBy("field")`, however, the lock argument represents an object held in a static field of another class. In our example, the inner `Helper` class's `counter` field uses the annotation to indicate to the users to acquire the `counterLock` of the `AnnotationsExampleClass` before manipulating the `counter` object.

`@GuardedBy("methodName()")#`

Indicates that the lock returned by the named method should be acquired before interacting with the annotated field or method. In our example class, the method `lockMaster()` is named in the annotation for the method `transforms()`. The annotation implies that `lockMaster(1)` should be invoked, the returned lock acquired and then only `transforms()` should be invoked.

@GuardedBy("ClassName.class")#

Indicates that the annotated field or method should be accessed only after acquiring the class literal object for the named class. In our example, the field `lastAccess` is annotated with the `@GuardedBy` annotation with the class object `AnnotationsExampleClass`. The method `updateAccessTime()` correctly acquires the lock on the class object before accessing the `lastAccess` field.

Summary#

The annotations we discussed in this lesson only serve to document notes and information for users and maintainers but can also be beneficial for static analysis tools that can programmatically verify if a class behaves as advertised. For instance, tools can verify if a class marked as `@Immutable` is truly immutable or not. In general, it is a good practise to annotate where appropriate for improving readability of one's code.

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {
    public static void main( String args[] ) throws Exception {
        ExecutorService threadPool = Executors.newFixedThreadPool(5);

        Callable<Void> someTask = new Callable<Void>() {

            public Void call() throws Exception {
                System.out.println("Running");
                return null;
            }
        };

        threadPool.submit(someTask).get();

        System.out.println( "Program Exiting" );
    }
}
```

The above program execution will show execution timed out, even though both the string messages are printed. You can fix the above code by adding `threadPool.shutdown()` as the last line of the method.

Question # 3

Which `compute()` method do you think would get invoked when `getWorking` is called?

```
class ThreadsWithLambda {

    public void getWorking() throws Exception {
        compute(() -> "done");
    }

    void compute(Runnable r) {
        System.out.println("Runnable invoked");
        r.run();
    }

    <T> T compute(Callable<T> c) throws Exception {
        System.out.println("Callable invoked");
        return c.call();
    }
}
```

The lambda expression is returning the string done, therefore the compiler will match the call to the second compute method and the expression will be considered a type of interface `Callable`. You can run the below snippet and verify the output to convince yourself.

```
import java.util.concurrent.Callable;

class Demonstration {
    public static void main( String args[] ) throws Exception{
        (new LambdaTargetType()).getWorking();
    }
}

class LambdaTargetType {

    public void getWorking() throws Exception {
        compute(() -> "done");
    }

    void compute(Runnable r) {
        System.out.println("Runnable invoked");
        r.run();
    }

    <T> T compute(Callable<T> c) throws Exception {
        System.out.println("Callable invoked");
        return c.call();
    }
}
```

Question # 4

What are the ways of representing tasks that can be executed by threads in Java?

Your Answer |

A)

Runnable interface and subclassing Thread class

B)

Passing anonymous class instance to thread constructor

C)

IComparable interface and subclassing Thread class

Question # 5

Given the code snippet below, how many times will the innerThread print its messages?

```
public void spawnThread() {  
    Thread innerThread = new Thread(new Runnable() {  
        public void run() {  
            for (int i = 0; i < 100; i++) {  
                System.out.println("I am a new thread !");  
            }  
        }  
    });  
    innerThread.start();  
    System.out.println("Main thread exiting");  
}
```

A)

innerThread prints a few messages and dies when the main thread exits

B)

innerThread prints exactly 100 messages even if the main thread exits before innerThread is done

C)

innerThread dies as soon as the main thread exits without printing any messages

Question # 6

Given the below code snippet how many messages will the innerThread print?

```
public void spawnDaemonThread() {  
    Thread innerThread = new Thread(new Runnable() {  
        public void run() {  
            for (int i = 0; i < 100; i++) {  
                System.out.println("I am a daemon thread !");  
            }  
        }  
    });  
    innerThread.setDaemon(true);  
    innerThread.start();  
    System.out.println("Main thread exiting");  
}
```

A)

exactly 100

B)

none

C)

a few

Question # 7

Say your program takes exactly 10 minutes to run. After reading this course, you become excited about introducing concurrency in your program. However, you only use two threads in your program. Holding all other variables constant, what is minimum time your improved program can theoretically run in?

A)

2 minutes

B)

5 minutes

C)

can't be determined.

D)

3.5 minutes

Question # 8

A sequential program is refactored to take advantage of threads. However, the programmer only uses two threads. The workload is divided such that one thread takes 9 times as long as the other thread to finish its work. What is the theoretical maximum speedup of the program as a percentage of the sequential running time?

A)

10%

B)

9%

C)

11%

D)

30%

Quiz 2

Questions on thread-safety and race conditions

Question # 1

What is a thread safe class?

A class is thread safe if it behaves correctly when accessed from multiple threads, irrespective of the scheduling or the interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.

Question # 2

Is the following class thread-safe?

```
public class Sum {  
    int sum(int... vals) {  
        int total = 0;  
        for (int i = 0; i < vals.length; i++) {  
            total += vals[i];  
        }  
        return total;  
    }  
}
```

A)

Yes

B)

No

[Reset Quiz](#)

[Submit Answer](#)

Show Explanation

The class **Sum** is stateless i.e. it doesn't have any member variables. All stateless objects and their corresponding classes are thread-safe. Since the actions of a thread accessing a stateless object can't affect the correctness of operations in other threads, stateless objects are thread-safe.

However, note that the method takes in variable arguments and the class wouldn't be thread safe anymore if the passed in argument was an array instead of individual integer variables and at the same the time, the **sum method performed a write operation on the passed in array.**

Question # 3

What is a race condition

A race condition occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime; in other words, getting the right answer relies on lucky timing. Two scenarios which can lead to a race condition are:

- **check-then-act:** Usually the value of a variable is checked and then an action is taken. Without proper synchronization, the resulting code can have a race condition. An example is below:

```
Object myObject = null;
if (myObject == null) {
    myObject = new Object();
}
```

- **read-modify-write:** For instance, whenever a counter variable is increment, the old state of the counter undergoes a transformation to a new state. Without proper synchronization guards, the counter increment operation can become a race condition.

Question # 4

Given the following code snippet, can you work out a scenario that causes a race condition?

```
1. class HitCounter {
2.
3.     long count = 0;
4.
5.     void hit() {
6.         count++;
7.     }
8.
9.     long getHits() {
10.        return this.count;
11.    }
}
```

The following sequence will result in a race condition.

1. Say **count = 7**
2. Thread A is about to execute line #6, which consists of fetching the **count** variable, incrementing it and then writing it back.
3. Thread A reads the count value equal to 7
4. Thread A gets context switched from the processor

5. Thread B executes line#6 atomically and increments **count = 8**
6. Thread A gets scheduled again
7. Thread A had previously read the **count = 7** and increment it to 8 and writes it back.
8. The net effect is **count** ends up with a value 8 when it should have been 9. This is an example of read-modify-write type of race condition.

Question # 5

Given the following code snippet, can you work out a scenario that causes a race condition?

```

1. class MySingleton {
2.
3.     MySingleton singleton;
4.
5.     private MySingleton() {
6.         }
7.
8.     MySingleton getInstance() {
9.         if (singleton == null)
10.             singleton = new MySingleton();
11.
12.         return singleton;
13.     }
14. }
```

This is the classic problem in Java for creating a singleton object. The following sequence will result in a race condition:

1. Thread A reaches line#9, finds the **singleton** object null and proceeds to line#10
2. Before executing line#10, Thread A gets context switched out
3. Thread B comes along and executes lines#9 and 10 atomically and the reference **singleton** is no more null.
4. Thread A gets scheduled on the processor again and new's up the **singleton** reference once more.
5. This is an example of a check-then-act use case that causes a race condition.

Quiz 3

Questions on how threads can be created

Question # 1

Give an example of creating a thread using the `Runnable` interface?

The below snippet creates an instance of the `Thread` class by passing in a lambda expression to create an anonymous class implementing the `Runnable` interface.

```
Thread t = new Thread(() -> {
    System.out.println(this.getClass().getSimpleName());
});

t.start();
t.join();
```

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

        Thread t = new Thread(() -> {
            System.out.println("Hello from thread !");
        });

        t.start();
        t.join();

    }
}
```

Question # 2

Give an example of a thread running a task represented by the `Callable<V>` interface?

There's no constructor in the `Thread` class that takes in a type of `Callable`. However, there is one that takes in a type of `Runnable`. We can't directly execute a callable task using an instance of the `Thread` class. However we can submit the callable task to an executor service. Both approaches are shown below:

Callable with Thread Class

```
// Anoymous class
Callable<Void> task = new Callable<Void>() {

    @Override
```

```

        public Void call() throws Exception {
            System.out.println("Using callable indirectly with instance of thread class");
            return null;
        }
    };

    // creating future task
    FutureTask<Void> ft = new FutureTask<>(task);
    Thread t = new Thread(ft);
    t.start();
    t.join();
}

```

Callable with Executor Service

```

// Anoymous class
Callable<Void> task = new Callable<Void>() {

    @Override
    public Void call() throws Exception {
        System.out.println("Using callable indirectly with instance of thread class");
        return null;
    }
};

ExecutorService executorService = Executors.newFixedThreadPool(5);
executorService.submit(task);
executorService.shutdown();
}

```

```

import java.util.concurrent.Callable;
import java.util.concurrent.FutureTask;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Demonstration {
    public static void main( String args[] ) throws Exception {
        usingExecutorService();
        usingThread();
    }

    static void usingExecutorService() {
        // Anoymous class
        Callable<Void> task = new Callable<Void>() {

            @Override
            public Void call() throws Exception {
                System.out.println("Using callable with executor service.");
                return null;
            }
        };
    }
}

```

```

ExecutorService executorService = Executors.newFixedThreadPool(5);
executorService.submit(task);
executorService.shutdown();
}

static void usingThread() throws Exception {
    // Anonymous class
    Callable<Void> task = new Callable<Void>() {

        @Override
        public Void call() throws Exception {
            System.out.println("Using callable indirectly with instance of thread class");
            return null;
        }
    };

    // creating future task
    FutureTask<Void> ft = new FutureTask<>(task);
    Thread t = new Thread(ft);
    t.start();
    t.join();

}
}

}

```

Question # 3

Give an example of representing a class using the [Thread class](#).

We can extend from the [Thread](#) class to represent our task. Below is an example of a class that computes the square roots of given numbers. The [Task](#) class encapsulates the logic for the task being performed.

```

class Task<T extends Number> extends Thread {

    T item;

    public Task(T item) {
        this.item = item;
    }

    public void run() {
        System.out.println("square root is: " + Math.sqrt(item.doubleValue
    }
}

```

```

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

        Thread[] tasks = new Thread[10];
        for(int i = 0;i<10;i++) {
            tasks[i] = new Task(i);
            tasks[i].start();
        }

        for(int i = 0;i<10;i++) {
            tasks[i].join();
        }
    }
}

class Task<T extends Number> extends Thread {

    T item;

    public Task(T item) {
        this.item = item;
    }

    public void run() {
        System.out.println("square root is: " + Math.sqrt(item.doubleValue()));
    }
}

```

Quiz 4

Question on use of synchronized

Question # 1

What is the **synchronized** keyword?

Java provides a built-in mechanism to provide atomicity called the **synchronized** block. A synchronized method is a shorthand for a synchronized block that spans an entire method body and whose lock is the object on which the method is being invoked.

A synchronized block consists of a reference to an object that serves as the lock and a block of code that will be guarded by the lock.

Synchronized blocks guarded by the same lock will execute one at a time. These blocks can be thought of as being executed atomically. Locks provide serialized access to the code paths they guard.

Below is an example of a class with a synchronized method.

```
class ContactBook {  
  
    Collection<String> contacts = new ArrayList<>();  
  
    synchronized void addName(String name) {  
        contacts.add(name);  
    }  
}
```

Note the synchronized method above is equivalent to the following rewrite:

```
void addName(String name) {  
    synchronized(this) {  
        contacts.add(name);  
    }  
}
```

Question # 2

Is the print statement in the below code reachable?

```
void doubleSynchronization() {  
  
    synchronized (this) {  
        synchronized (this) {  
            System.out.println("Is this line unreachable ?");  
        }  
    }  
}
```

A)

Code isn't reachable because we try to synchronize twice on the same object

B)

Code is reachable because we can synchronize multiple times on the same object

Show Explanation

Question # 3

Consider the below class which has a synchronized method. Can you tell what object does the thread invoking the `addName()` method synchronize on?

```
class ContactBook {  
    Collection<String> contacts = new ArrayList<>();  
  
    synchronized void addName(String name) {  
        contacts.add(name);  
    }  
}
```

Class may be used as follows:

```
ContactBook contactBook = new ContactBook();  
contactBook.addName("Trump");
```

A)

The main thread

B)

Thread invoking the method

C)

The object on which a thread invokes the method

D)

synchronize is a synchronization keyword and doesn't require any objects for operation

Hide Explanation

The method is synchronized on the object on which a thread invokes the method. In the example usage, the method will be synchronized on the `contactsBook` object. Also note that prefixing the method signature with `synchronized` is equivalent of the defining the method in the following manner:

```
void addName(String name) {  
    synchronized (this) {  
        contacts.add(name);  
    }  
}
```

Question # 4

An instance method synchronizes on the instance object, do you know what object do static methods synchronize on?

Hide Explanation

A static synchronized method synchronizes on the **class object**.

Quiz 5

Exercise on how to make classes thread-safe

Question # 1

Is the following class thread-safe?

```
public class Sum {  
    int count = 0;  
    int sum(int... vals) {  
        count++;  
  
        int total = 0;  
        for (int i = 0; i < vals.length; i++) {  
            total += vals[i];  
        }  
        return total;  
    }  
  
    void printInvocations() {  
        System.out.println(count);  
    }  
}
```

A)

Yes

B)

No

[Reset Quiz](#)

[Submit Answer](#)

Hide Explanation

The class isn't thread-safe because it has state that can be mutated by different threads without synchronization amongst them. The state consists of the variable **count**

Question # 2

What are the different ways in which we can make the [Sum](#) class thread-safe?

We can use an instance of the [AtomicInteger](#) for keeping the count of invocations. The thread-safe code will be as follows:

Using Atomic Integer

```
public class SumFixed {  
    AtomicInteger count = new AtomicInteger(0);  
  
    int sum(int... vals) {  
        count.getAndIncrement();  
  
        int total = 0;  
        for (int i = 0; i < vals.length; i++) {  
            total += vals[i];  
        }  
        return total;  
    }  
  
    void printInvocations() {  
        System.out.println(count.get());  
    }  
}
```

We can also fix the sum class by using synchronizing on the object instance.

Using Synchronization on [this](#)

```
public class SumFixed {  
    int count = 0;  
  
    synchronized int sum(int... vals) {  
        count++;  
  
        int total = 0;
```

```

        for (int i = 0; i < vals.length; i++) {
            total += vals[i];
        }
        return total;
    }

    synchronized void printInvocations() {
        System.out.println(count);
    }
}

```

We could also use another object other than `this` for synchronization. The code would then be as follows:

```

public class SumFixed {

    int count = 0;
    Object lock = new Object();

    int sum(int... vals) {
        synchronized (lock) {
            count++;
        }

        int total = 0;
        for (int i = 0; i < vals.length; i++) {
            total += vals[i];
        }
        return total;
    }

    void printInvocations() {
        synchronized (lock) {
            System.out.println(count);
        }
    }
}

```

Question # 3

In the above question, when we fixed the `Sum` class for thread safety we synchronized the `printInvocations()` method. What will happen if we didn't synchronize the `printInvocations()` method?

The `printInvocations()` method performs a read-only operation of the shared variable `count`. If we skipped synchronizing the method, then the method call can potentially return/print stale value for the `count` variable including zero.

One may be tempted to skip synchronizing the read-only access of variables if the application logic can tolerate stale values for a variable but that is a

dangerous proposition. Writes to the `count` variable may not be visible to other threads because of how the Java's memory model works. We'll need to declare the `count` variable `volatile` to ensure threads reading it see the most recent value. However, marking a variable `volatile` will not eliminate race conditions.

Question # 4

If we synchronize the `sum()` method as follows, will it be thread-safe?

```
int sum(int... vals) {  
  
    Object myLock = new Object();  
    synchronized (myLock) {  
        count++;  
    }  
  
    int total = 0;  
    for (int i = 0; i < vals.length; i++) {  
        total += vals[i];  
    }  
    return total;  
}
```

A)

Yes

B)

No

[Reset Quiz](#)

[Submit Answer](#)

Hide Explanation

If multiple threads call into the `sum()` method then each thread will create a `myLock` object on its thread stack on which it will synchronize. In order to ensure thread-safety, threads need to synchronize on the same object.

Quiz 6

Questions regarding memory visibility in multithreaded scenarios

Question # 1

Consider the below class:

```
public class MemoryVisibility {  
    int myvalue = 2;  
    boolean done = false;  
  
    void thread1() {  
        while (!done);  
        System.out.println(myvalue);  
    }  
  
    void thread2() {  
        myvalue = 5;  
        done = true;  
    }  
}
```

We create an object of the above class and have two threads run each of the two methods like so:

```
MemoryVisibility mv = new MemoryVisibility();  
  
Thread thread1 = new Thread(() -> {  
    mv.thread1();  
});  
  
Thread thread2 = new Thread(() -> {  
    mv.thread2();  
});  
  
thread1.start();  
thread2.start();  
  
thread1.join();  
thread2.join();
```

What will be the output by thread1?

A)

loops forever and doesn't print anything

B)

prints 5

C)

prints 2

D)

May loop forever or print 2 or print 5

[Reset Quiz](#)

[Submit Answer](#)

Hide Explanation

This is a classic *gotcha* moment for a newbie to Java concurrency. Remember in the absence of synchronization, the compiler, processor or the runtime are free to take liberty in reordering operations. There's no guarantee that the values written to `myvalue` and `done` by thread2 are visible to thread1 in the same order or visible at all. The updated values may reside in the processor cache and not be immediately propagated to main memory from where thread1 reads. It's also possible that thread1 sees updated value for one of the variables and not for another. Synchronization is not only about mutual exclusion but also about memory visibility.

Question # 2

Will the following change guarantee that thread1 sees the changes made to shared variables by thread2?

```
public class MemoryVisibility {  
    int myvalue = 2;  
    boolean done = false;  
  
    void thread1() {  
        synchronized (this) {  
            while (!done);  
            System.out.println(myvalue);  
        }  
    }  
  
    void thread2() {  
        myvalue = 5;  
        done = true;  
    }  
}
```

A)

Yes

B)

No

[Reset Quiz](#)

[Submit Answer](#)

Hide Explanation

This is a prime example of insufficient synchronization. The reader thread may still see stale values of the shared variables as they may be updated by writer but only in a register or cache.

Question # 3

Does `synchronized` ensure memory visibility? How can we fix the above code using synchronization?

We have already seen that synchronization ensures **atomicity** i.e. operations within a synchronized code block all execute together without interruption. You can imagine these operations to be executed like a transaction, where either all of them execute or none execute.

From a memory visibility perspective, say two threads A and B are synchronized on the same object. Once thread A exits the synchronized block (releases the lock), all the variable values that were visible to thread A prior to leaving the synchronized block (releasing the lock) will become visible to thread B as soon as thread B enters the synchronized block (acquires the lock).

The memory visibility class can be fixed with synchronization as follows:

```
public class MemoryVisibility {  
  
    int myvalue = 2;  
    boolean done = false;  
  
    void thread1() throws InterruptedException {  
  
        synchronized (this) {  
            while (!done)  
                this.wait();  
            System.out.println(myvalue);  
        }  
    }  
}
```

```

void thread2() {
    synchronized (this) {
        myvalue = 5;
        done = true;
        this.notify();
    }
}

```

Question # 4

Describe `volatile`? Can it help us with the `MemoryVisibility` class.

When a field is declared `volatile`, it is an indication to the compiler and the runtime that the field is shared and operations on it shouldn't be reordered. Volatile variables aren't cached in registers or caches where they are hidden from other processors. Note that variables declared `volatile` when read always return the most recent write by any thread.

Furthermore volatile variables only guarantee memory visibility but not atomicity.

In the fixed `MemoryVisibility` class using synchronization may seem an overkill as acquiring and releasing locks is never cheap. Volatile provides a weaker form of synchronization and can alleviate the situation in the `MemoryVisibility` class if we declare both the shared variables `volatile`.

Question # 5

Will it be enough to declare the `done` flag `volatile` or do we need to declare `myvalue` `volatile` too?

```

public class MemoryVisibility {

    int myvalue = 2;
    volatile boolean done = false;

    void thread1() {
        while (!done);
        System.out.println(myvalue);
    }

    void thread2() {
        myvalue = 5;
        done = true;
        this.notify();
    }
}

```

```
}
```

It is intuitive to think that if we declare just the boolean flag `volatile`, it'll prevent from infinite looping but the latest value for the variable `myvalue` may not get printed, since it is not declared `myvalue`. However, that is not true and we can get away by only declaring the boolean flag as `volatile`. Though note that declaring both the shared variables `volatile` is acceptable too.

Writing to a `volatile` variable is akin to exiting a synchronized block and reading a volatile variable is akin to entering a synchronized variable. Similar to the visibility guarantees for a synchronized block, after a reader-thread reads a volatile variable, it sees the same values of all the variables as seen by a writer-thread just before the writer-thread wrote to the same volatile variable.

Question # 6

If we introduced a third thread that could also mutate the value of `myvalue` variable in the fixed `MemoryVisibility` class that uses `volatile`, how can that affect the value printed by `thread1`?

Consider the below sequence of thread scheduling

- Thread 2 changes the value of `myvalue` to 5 and sets the volatile flag `done` to true
- Thread 3 mutates the value of `myvalue` to say 16 that gets stored in the register
- Thread 1 when scheduled will be guaranteed to see all the values of variables when `done` was updated to true by thread 2. At that time `myvalue` was set to 5 and even though thread 3 changed it to 16, there's no guarantee that thread 2 sees it because it happened after the write to the volatile variable. Therefore at this point it may print 5 or 16.

Question # 7

When is `volatile` most commonly used?

Common situation where `volatile` can be used are:

- Most common use of volatile variables is as a interruption, completion or status flag

- When writes to a variables don't depend on its current value e.g. a counter is not suitable to be declared volatile as its next value depends on its current value.
- When a single thread ever writes to the variable. Imagine a scenario where only a single thread writes or modifies a shared volatile variable but the variable is read by several other threads. In this situation, race conditions are prevented because only one thread is allowed to write to the shared variable and visibility guarantees of volatile ensure other threads see the most up to date value.
- When locking isn't required for reading the variable or that the variable doesn't participate in maintaining a variant with other state variables

Quiz 7

Threaded design and thread-safety questions.

Question # 1

Can you enumerate the implications of the poor design choice for the below class?

```
public class BadClassDesign {
    private File file;

    public BadClassDesign() throws InterruptedException {
        Thread t = new Thread(() -> {
            System.out.println(this.file);
        });
        t.start();
        t.join();
    }
}
```

The above class is a bad design choice for the following reasons:

- When creating the thread object in the constructor, the reference to the instance of the enclosing `BadClassDesign` class is also implicitly captured by the anonymous class that implements `Runnable`. The problem with this approach is that the anonymous class can attempt to use the enclosing object while it is still being constructed. This would not be an issue if we didn't start the thread in the constructor. Note that if we invoked an overrideable instance method in the

constructor, we'll be giving a derived class a chance to access the half constructed object in an unsafe manner.

- The private fields of the **BadClassDesign** class also become accessible to the instance of the anonymous inner class that we pass in to the **Thread** class's constructor.

```
import java.io.File;

class Demonstration {
    public static void main( String args[] ) throws Exception {
        BadClassDesign bcd = (new BadClassDesign());
    }
}

class BadClassDesign {

    // Private field
    private File file;

    public BadClassDesign() throws InterruptedException {
        Thread t = new Thread(() -> {
            System.out.println(this.getClass().getSimpleName());

            // Private field of class is accessible in the anonymous class
            System.out.println(this.file);
        });
        t.start();
        t.join();
    }
}
```

Question # 2

What is stack-confinement in the context of threading?

All local variables live on the executing thread's stack and are confined to the executing thread. This intrinsically makes a snippet of code thread-safe. For instance consider the following instance method of a class:

```
int getSum(int n) {

    int sum = 0;
    for (int i = 1; i <= n; i++)
        sum += i;
    return sum;
```

```
}
```

If several threads were to simultaneously execute the above method, the execution by each thread would be thread-safe since all the threads will have their own copies of the variables in the method above.

Primitive local types are always stack confined but care has to be exercised when dealing with local reference types as returning them from methods or storing a reference to them in shared variables can allow simultaneous manipulation by multiple threads thus breaking stack confinement.

Quiz 8

Questions on working with ThreadLocal variables

Question # 1

Consider the class below:

```
public class Counter {  
  
    ThreadLocal<Integer> counter = ThreadLocal.withInitial(() -> 0);  
  
    public Counter() {  
        counter.set(10);  
    }  
  
    void increment() {  
        counter.set(counter.get() + 1);  
    }  
}
```

What would be the output of the method below when invoked?

```
public void usingThreads() throws Exception {  
  
    Counter counter = new Counter();  
    Thread[] tasks = new Thread[100];  
  
    for (int i = 0; i < 100; i++) {  
        Thread t = new Thread(() -> {  
            for (int j = 0; j < 100; j++)  
                counter.increment();  
        });  
        tasks[i] = t;  
        t.start();  
    }  
  
    for (int i = 0; i < 100; i++) {
```

```
        tasks[i].join();
    }

    // What is the output of the the below line?
    System.out.println(counter.counter.get());
}
```

A)

O

B)

10

C)

100

D)

110

E)

10000

[Reset Quiz](#)

[Submit Answer](#)

Hide Explanation

Note that aside from the 100 threads that we create, there's the **main** thread that creates those 100 threads. The constructor for the **counter** variable is invoked only by the main thread and its copy of the threadlocal variable is initialized to 10. The main thread never increments the variable so the print statement prints out 10.

Note that we must initialize the ThreadLocal variable inline than in the constructor because the constructor would only be invoked for a single thread. Usually, static variables are declared threadlocal to keep separate values on a per thread basis.

Question # 2

Given the same Counter class as in the previous question, what is the output of println statement below:

```
public void usingSingleThreadPool() throws Exception {  
    Counter counter = new Counter();  
    ExecutorService es = Executors.newFixedThreadPool(1);  
    Future<Integer>[] tasks = new Future[100];  
  
    for (int i = 0; i < 100; i++) {  
        tasks[i] = es.submit(() -> {  
            for (int j = 0; j < 100; j++)  
                counter.increment();  
  
            return counter.counter.get();  
        });  
    }  
  
    // What is the output of the below line?  
    System.out.println(tasks[99].get());  
  
    es.shutdown();  
}
```

A)

O

B)

10

C)

100

D)

10000

[Reset Quiz](#)

[Submit Answer](#)

Hide Explanation

This example may seem counterintuitive but drives home the warning of using threadlocal variables with threadpools! Note that even though we submit 100 tasks but our threadpool consists of a lone thread. This thread has a copy of threadlocal variable **counter** which gets reused when executing different tasks. It is not issued a new copy of the variable for every task as one might mistakenly assume. The same copy gets incremented for all 100 tasks a 100 times resulting in a value of 10,000 being printed.

Question # 3

What would have been the output of the print statement from the previous question if we created a pool with 20 threads?

A)

0

B)

10

C)

100

D)

1000

E)

between 100 and 10000 inclusive

[Reset Quiz](#)

[Submit Answer](#)

Hide Explanation

Since our threadpool consists of 20 threads the print statement may print a value between the two extremes. If only a single thread gets to execute all the tasks then counter will print a maximum value of 10,000. If a thread only executed the 100th task and all the other 99 tasks were performed by the remaining 19 threads then the printed value would be 100.

The code for all the three scenarios discussed above appears below.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

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

        usingThreads();
        usingSingleThreadPool();
    }
}
```

```
usingMultiThreadsPool();
}

static void usingThreads() throws Exception {

    Counter counter = new Counter();
    Thread[] tasks = new Thread[100];

    for (int i = 0; i < 100; i++) {
        Thread t = new Thread(() -> {
            for (int j = 0; j < 100; j++)
                counter.increment();
        });
        tasks[i] = t;
        t.start();
    }

    for (int i = 0; i < 100; i++) {
        tasks[i].join();
    }

    System.out.println(counter.counter.get());
}

@SuppressWarnings("unchecked")
static void usingSingleThreadPool() throws Exception {

    Counter counter = new Counter();
    ExecutorService es = Executors.newFixedThreadPool(1);
    Future<Integer>[] tasks = new Future[100];

    for (int i = 0; i < 100; i++) {
        tasks[i] = es.submit(() -> {
            for (int j = 0; j < 100; j++)
                counter.increment();

            return counter.counter.get();
        });
    }

    System.out.println(tasks[99].get());

    es.shutdown();
}

@SuppressWarnings("unchecked")
static void usingMultiThreadsPool() throws Exception {
```

```

Counter counter = new Counter();
ExecutorService es = Executors.newFixedThreadPool(20);
Future<Integer>[] tasks = new Future[100];

for (int i = 0; i < 100; i++) {
    tasks[i] = es.submit(() -> {
        for (int j = 0; j < 100; j++)
            counter.increment();

        return counter.counter.get();
    });
}

System.out.println(tasks[99].get());

es.shutdown();
}

}

class Counter {

    ThreadLocal<Integer> counter = ThreadLocal.withInitial(() -> 0);

    public Counter() {
        counter.set(10);
    }

    void increment() {
        counter.set(counter.get() + 1);
    }
}

```

Question # 4

Consider the below method:

```

int countTo100() {

    ThreadLocal<Integer> count = ThreadLocal.withInitial(() -> 0);
    for (int j = 0; j < 100; j++)
        count.set(count.get() + 1);

    return count.get();
}

```

The above code is invoked like so:

```
ExecutorService es = Executors.newFixedThreadPool(1);
Future<Integer>[] tasks = new Future[100];

for (int i = 0; i < 100; i++) {
    tasks[i] = es.submit(() -> countTo100());
}

for (int i = 0; i < 100; i++)
    System.out.println(tasks[i].get());

es.shutdown();
```

What would the output of the print statement for the 100 tasks?

A)

100

B)

10000

C)

between 100 and 10,000 inclusive

[Reset Quiz](#)

[Submit Answer](#)

Hide Explanation

Note that the threadlocal variable is now within the instance method. Even though we have a single thread but each time it invokes the `countTo100()` method, it creates a fresh threadlocal object which has no relation to the threadlocal object from the previous invocation. The scope of the threadlocal variable is limited to within the instance method and as soon as the thread exits the method, it is eligible for garbage collection. On the next invocation a new threadlocal object is created.

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {

    @SuppressWarnings("unchecked")
    public static void main( String args[] ) throws Exception {

        ExecutorService es = Executors.newFixedThreadPool(1);
        Future<Integer>[] tasks = new Future[100];

        for (int i = 0; i < 100; i++) {
            tasks[i] = es.submit(() -> countTo100());
        }

        for (int i = 0; i < 100; i++)
            System.out.println(tasks[i].get());

        es.shutdown();
    }

    static int countTo100() {

        ThreadLocal<Integer> count = ThreadLocal.withInitial(() -> 0);
        for (int j = 0; j < 100; j++)
            count.set(count.get() + 1);

        return count.get();
    }
}

```

Question # 5

Is there any benefit to declaring `count` as a threadlocal variable in the method `countTo100()`?

```

int countTo100() {

    ThreadLocal<Integer> count = ThreadLocal.withInitial(() -> 0);
    for (int j = 0; j < 100; j++)
        count.set(count.get() + 1);

    return count.get();
}

```

The variables defined inside an instance method are already created on a per-thread basis and live on the thread stack without any sharing with other threads. The per-thread level isolation for a variable that we can achieve using `threadlocal` is already being provided because of the scope of the variables declared within an instance method. Therefore, there's no benefit to declaring variables within instance methods as `threadlocal`.