

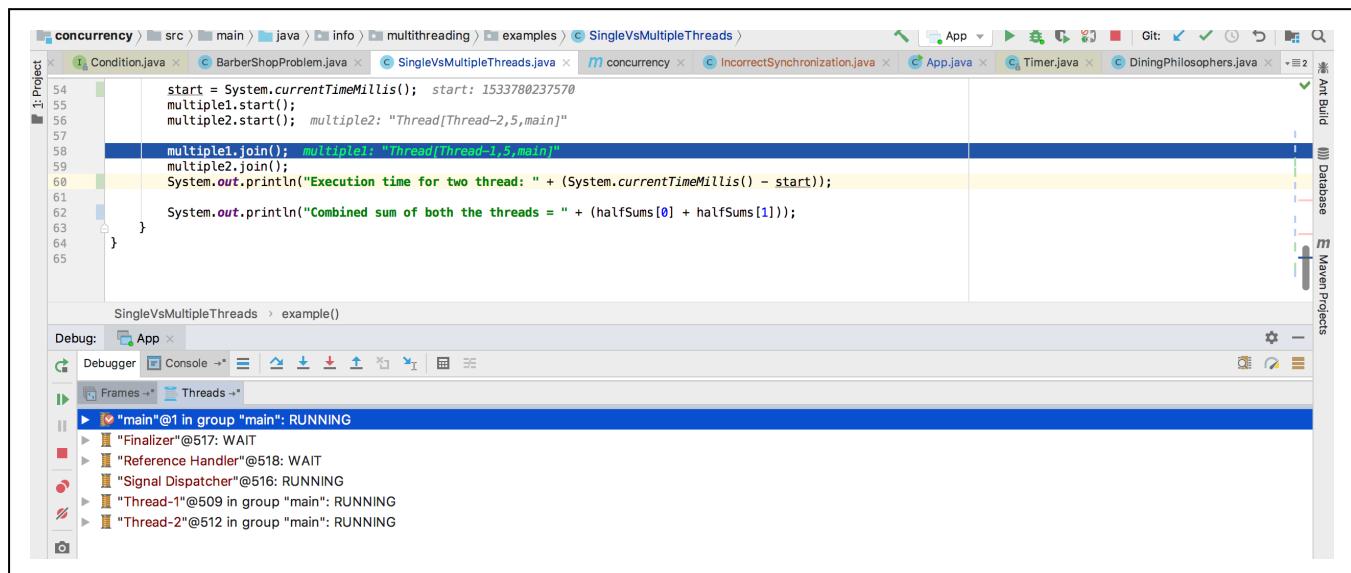
# Introduction

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

## Introduction

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.



The screenshot shows an IDE interface with a code editor and a debugger window. The code editor displays Java code for a thread example. The debugger window shows a list of threads, with several threads labeled as 'RUNNING'.

```
54 start = System.currentTimeMillis(); start: 1533780237570
55 multiple1.start();
56 multiple2.start(); multiple2: "Thread[Thread-2,5,main]"
57
58 multiple1.join(); multiple1: "Thread[Thread-1,5,main]"
59 multiple2.join();
60 System.out.println("Execution time for two thread: " + (System.currentTimeMillis() - start));
61
62 System.out.println("Combined sum of both the threads = " + (halfSums[0] + halfSums[1]));
63
64 }
65 }
```

Suspended threads in a debugger

The simplest example to think of a concurrent system is a single

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 0 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.

```
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();

    t1.join();
    t2.join();

    long finalCount = s1.counter + s2.counter;
    long end = System.currentTimeMillis();
    System.out.println("Two threads final count = " + finalCount + " took " + (end - start));
}

static public void oneThread() {

    long start = System.currentTimeMillis();
    SumUpExample s = new SumUpExample(1, MAX_NUM );
    s.add();
    long end = System.currentTimeMillis();
    System.out.println("Single thread final count = " + s.counter + " took " + (end - start));
}
}

public static void runTest() throws InterruptedException {
    oneThread();
    twoThreads();
}
}

```



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.

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.

## 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

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.

## Notes

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.

## Process

### Global Variables



#### Thread

local Variables

Code

#### Thread

local Variables

Code

#### Thread

local Variables

Code

## 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 {
        int counter = 0;
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000000; i++) {
                counter++;
            }
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000000; i++) {
                counter--;
            }
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(counter);
    }
}
```

```

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);
    }
}

```

```
    }  
}
```



### Example of Thread Unsafe Code

# Concurrency vs Parallelism

This lesson clarifies the common misunderstandings and confusions around concurrency and 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 one part of a program may overlap with the execution of another part.

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.

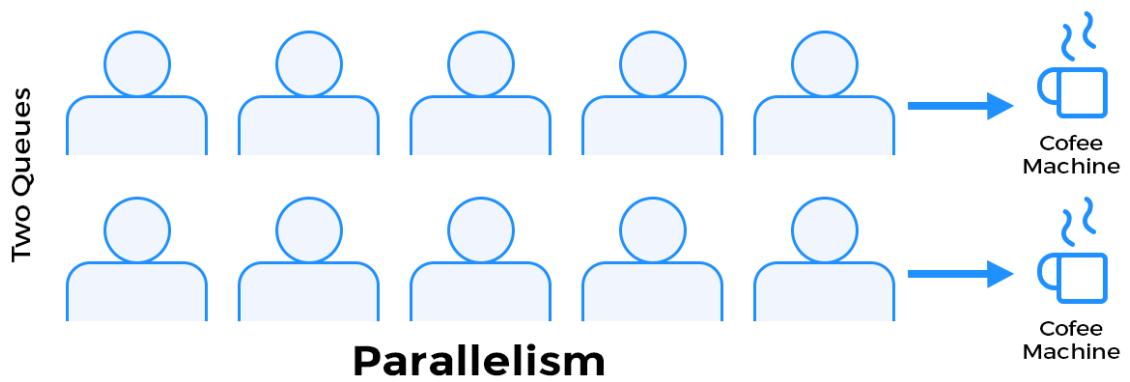
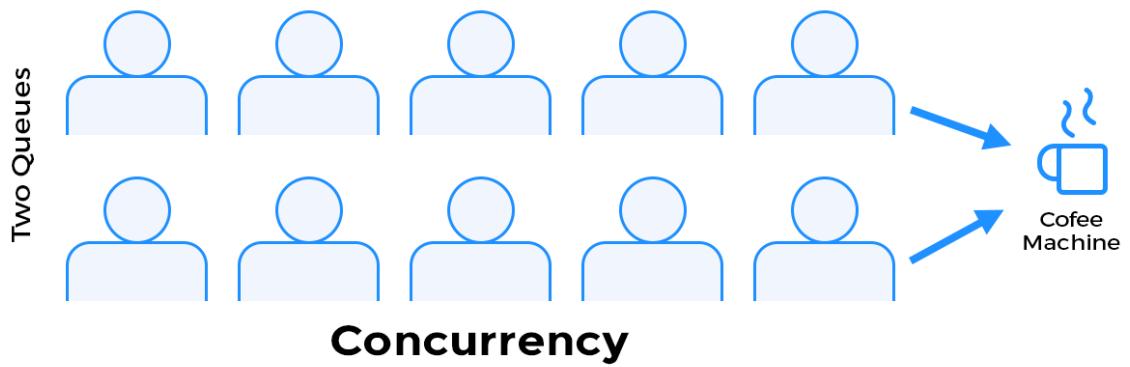
Revisiting our juggler analogy, a parallel system would map to at least two or more jugglers juggling one or more balls. In the case of an operating system, if it runs on a machine with say four CPUs then the operating system can execute four tasks at the same time, making execution parallel. Either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

## 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.

## 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.

# Synchronous vs Asynchronous

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

## 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.

## 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 compilers, optimizers, and scientific applications.

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.

## Notes

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.

## 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 different files. Of course, there will be some overhead due to

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.

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 verison 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();
}
}

```



### Race Condition Example

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.

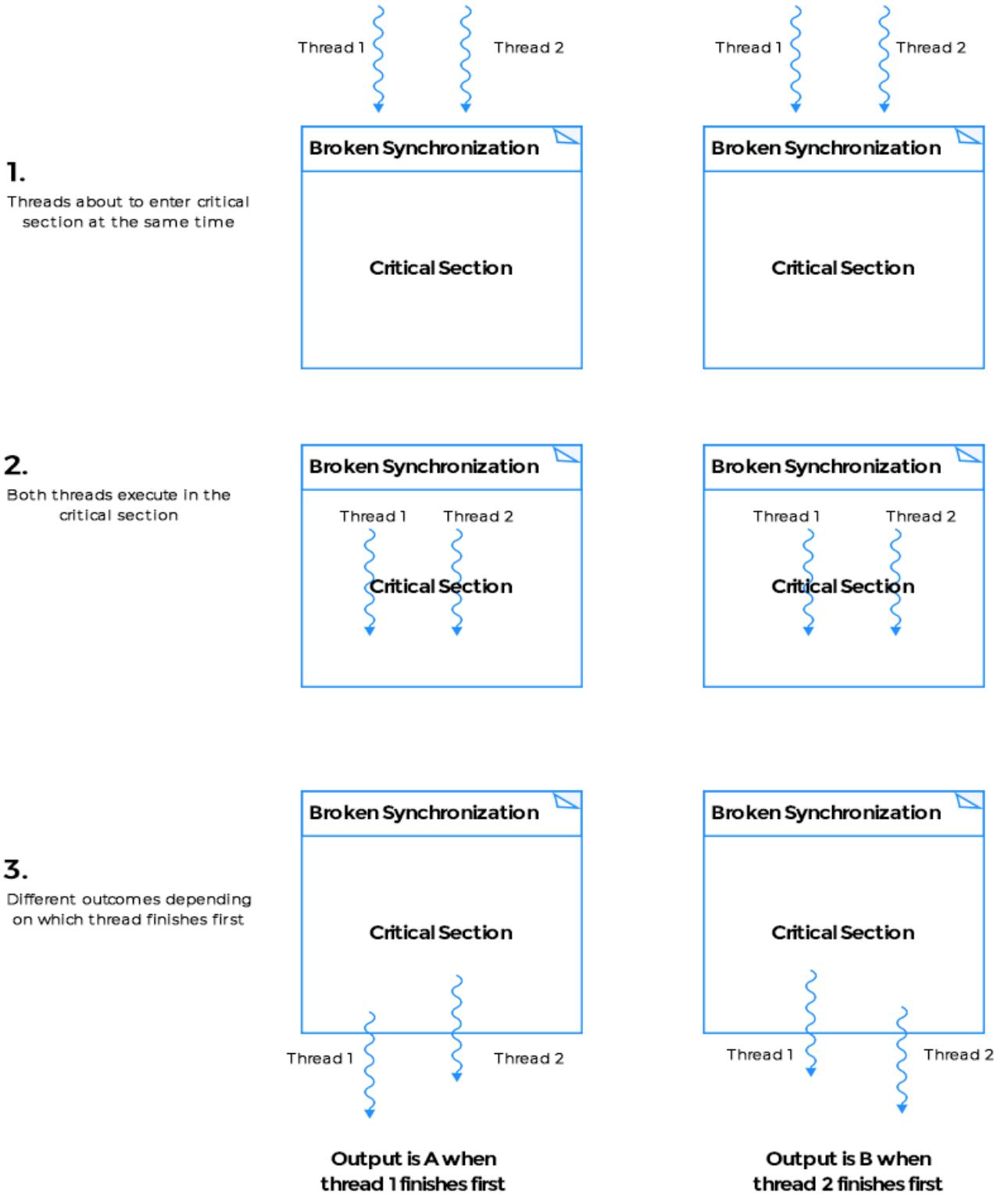
For the impatient, the fix is presented below where we guard the read and write of the **randInt** variable using the **RaceCondition** object as the monitor. Don't fret if the solution doesn't make sense for now, it would, once we cover various topics in the lessons ahead.

```
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() {
```

```
    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.

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 other's actions, leading to an indefinite loop. This happens when two threads are waiting for each other to release a lock.

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 text-book 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) {  
        }  
    }  
}
```



```
        } 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();
                    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--;
            }
        }
    }
}
```



### Example of a Deadlock

## 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();
    }
}
```



Example of Deadlock with Non-Reentrant Lock

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!

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

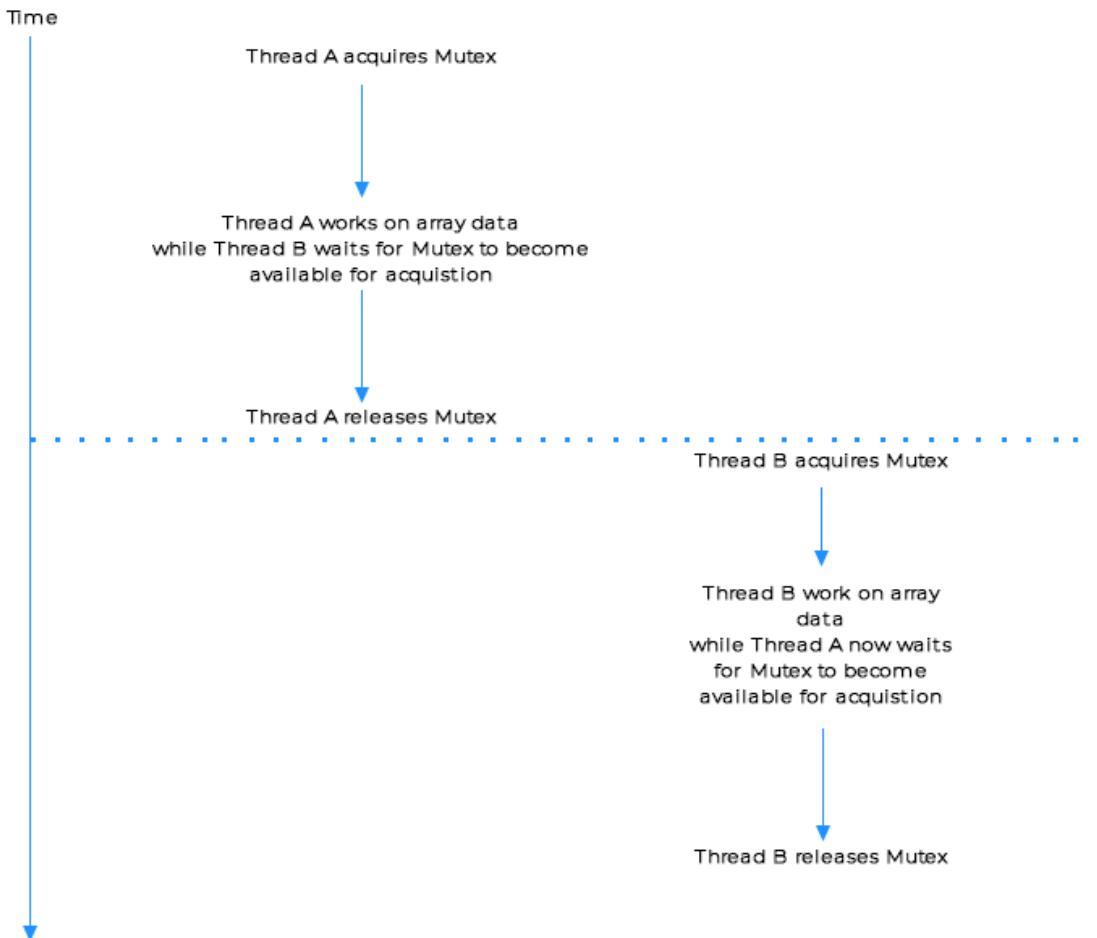
Semaphore, on the other hand, is used for limiting access to a collection of resources. Think of semaphore as having a limited number of permits to give out. If a semaphore has given out all the permits it has, then any new thread that comes along requesting for a permit will be blocked, till an

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.

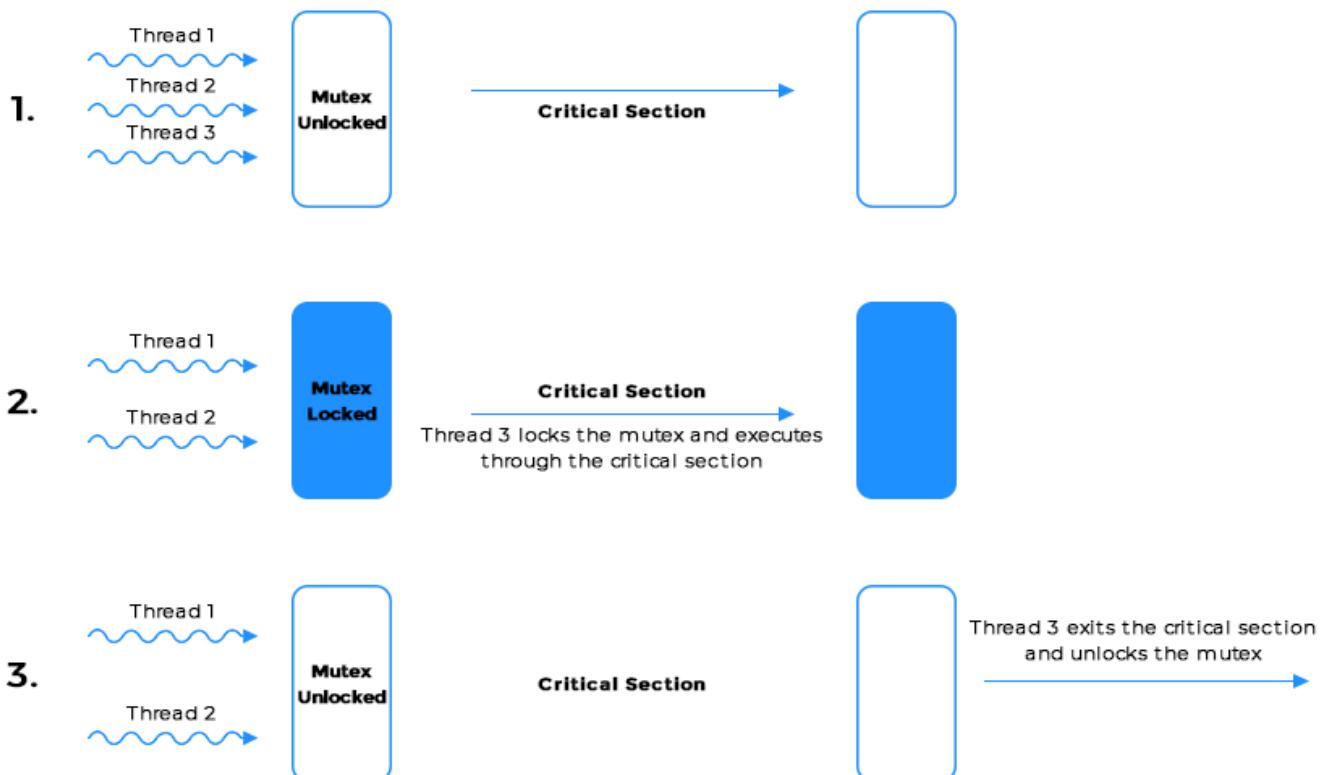
If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlock

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.

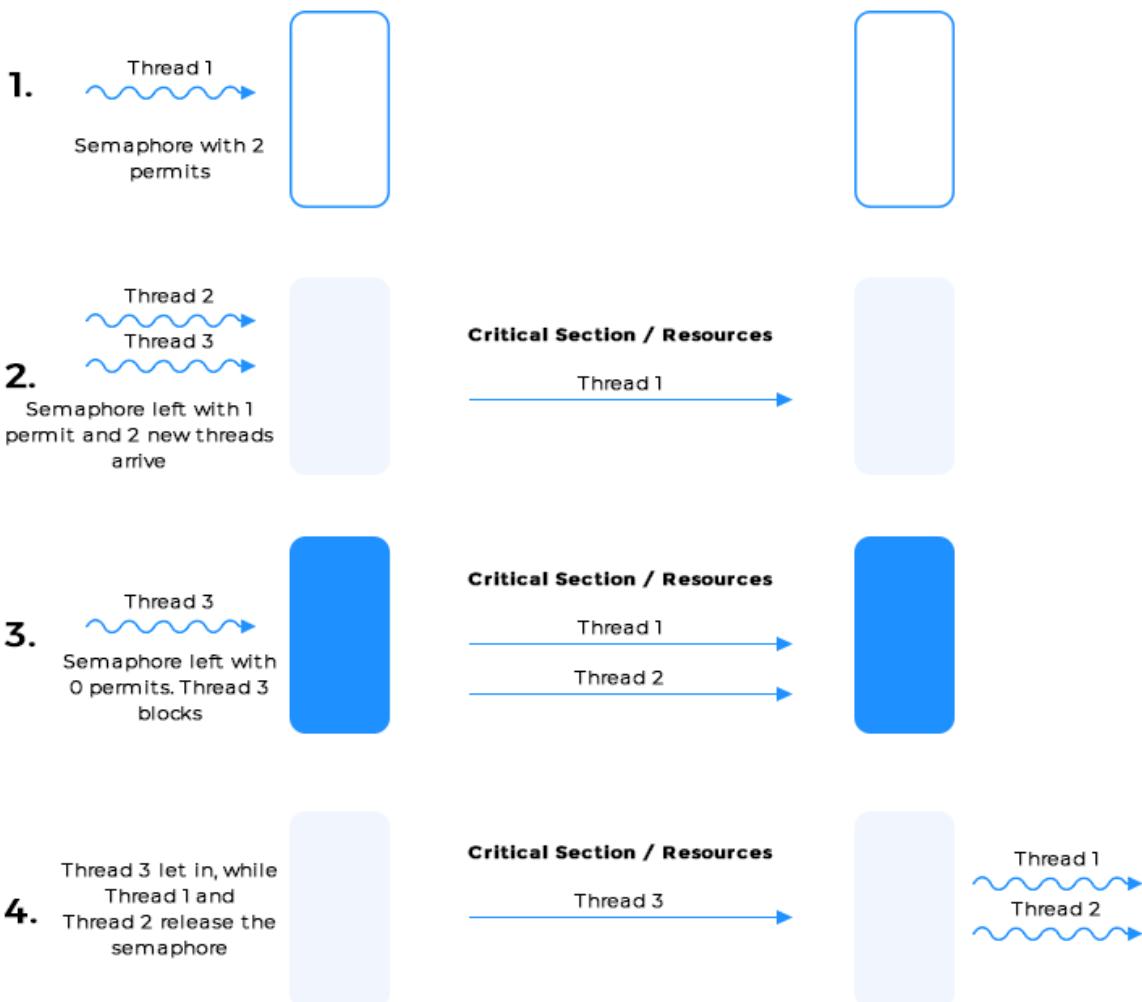
## Semaphore for Signaling

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.

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** 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

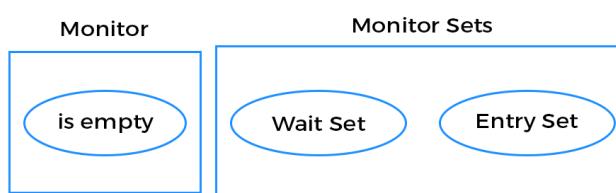
**cooperation** by allowing them to wait and signal on conditions.

Initial Monitor State



A pictorial representation of the above simulation appears below:

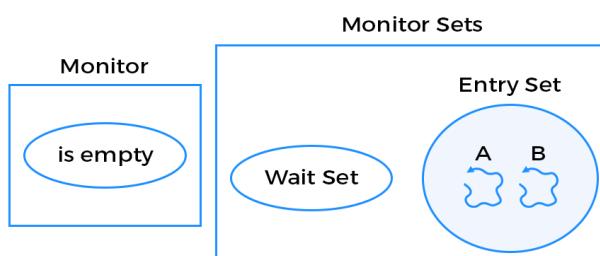
**1. Initial Monitor State**



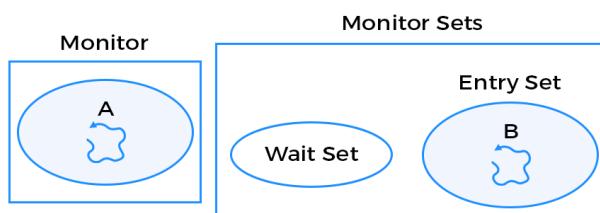
**2. Two threads come along to enter the monitor**



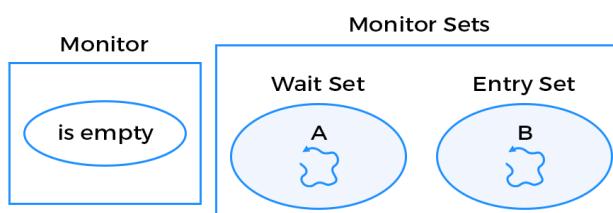
**3. Thread A and B get placed in the entry set**



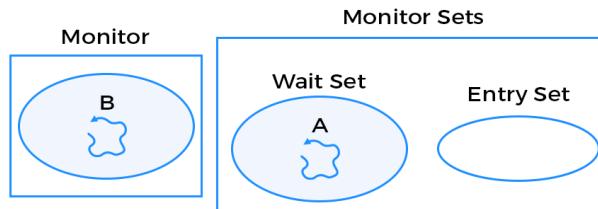
**4. Thread A enters the monitor and starts execution**



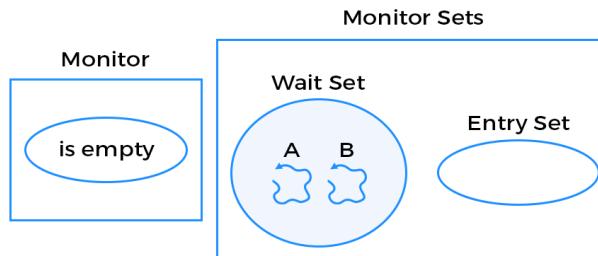
**5. Thread A execution Wait() and gets placed in wait set**



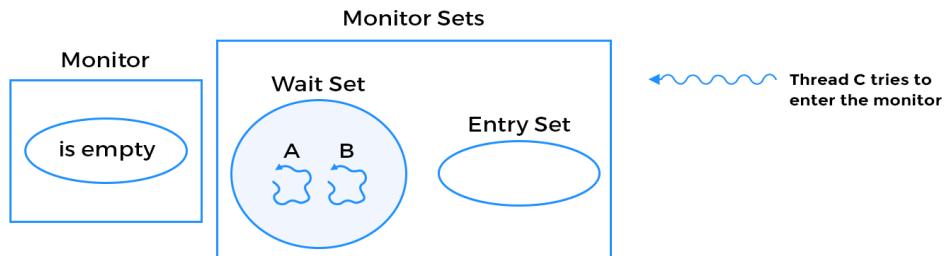
6. Thread B now able to enter the monitor



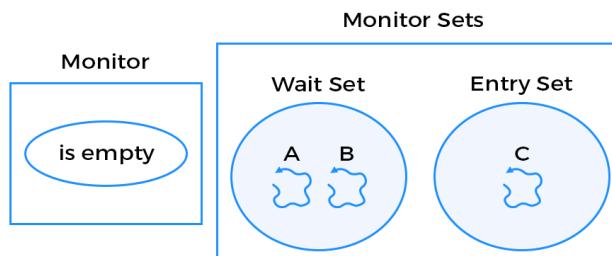
7. Thread B also invokes wait() and gets placed in the wait set



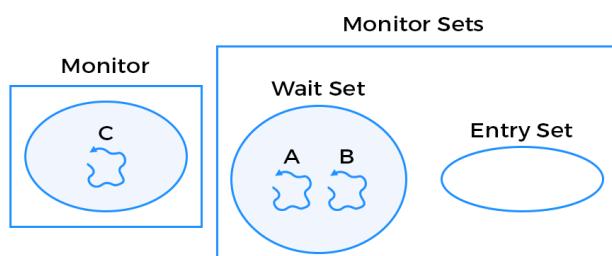
8. Thread C comes along to enter the monitor



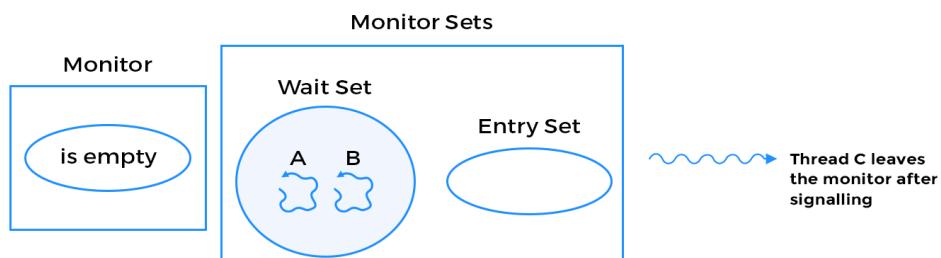
9. Thread C is placed in the entry set



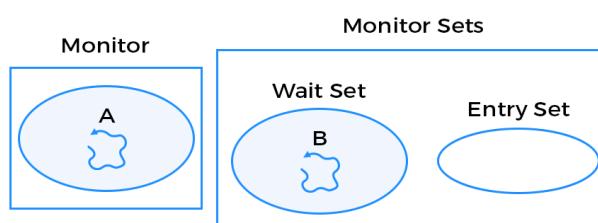
10. Thread C enters the monitor



11. Thread C exits monitor after signalling



12. Thread A resumes ownership of 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 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();  
    }  
}
```



Illegal Monitor Exception

## 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.

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.

## The Difference

- 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

## 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

- $(1 - P)$  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:

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

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.

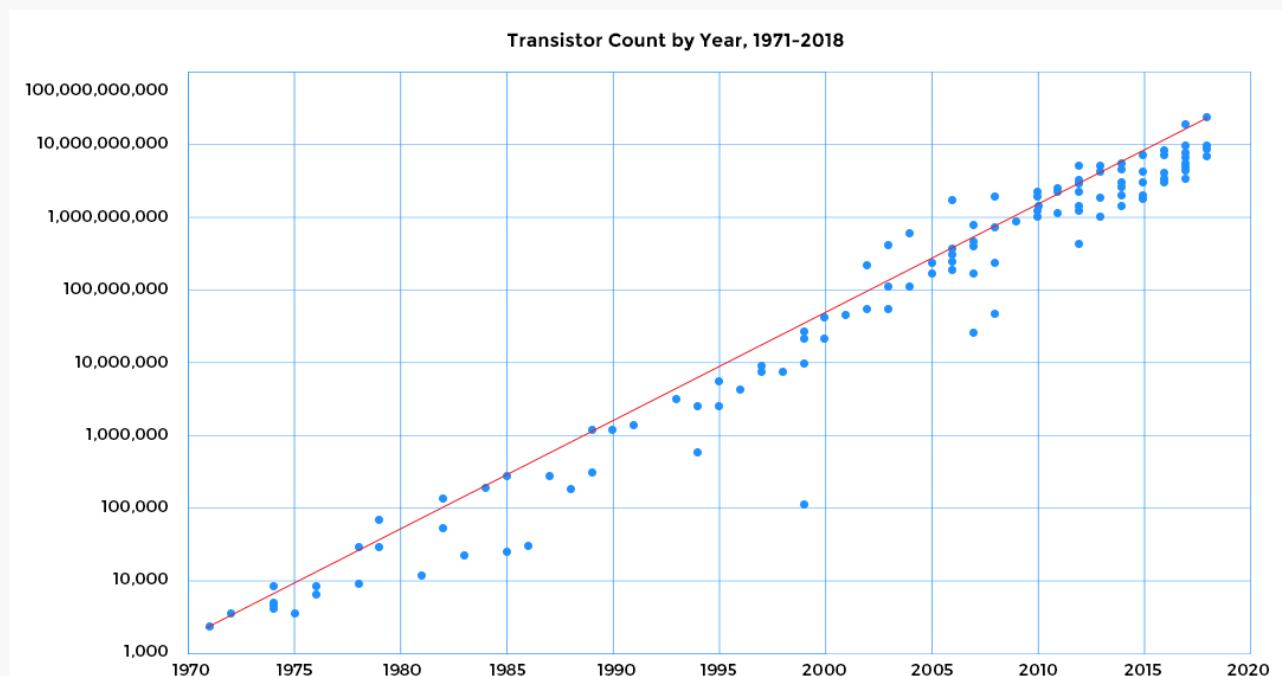
# Moore's Law

Discusses impact of Moore's law on concurrency.

## 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.

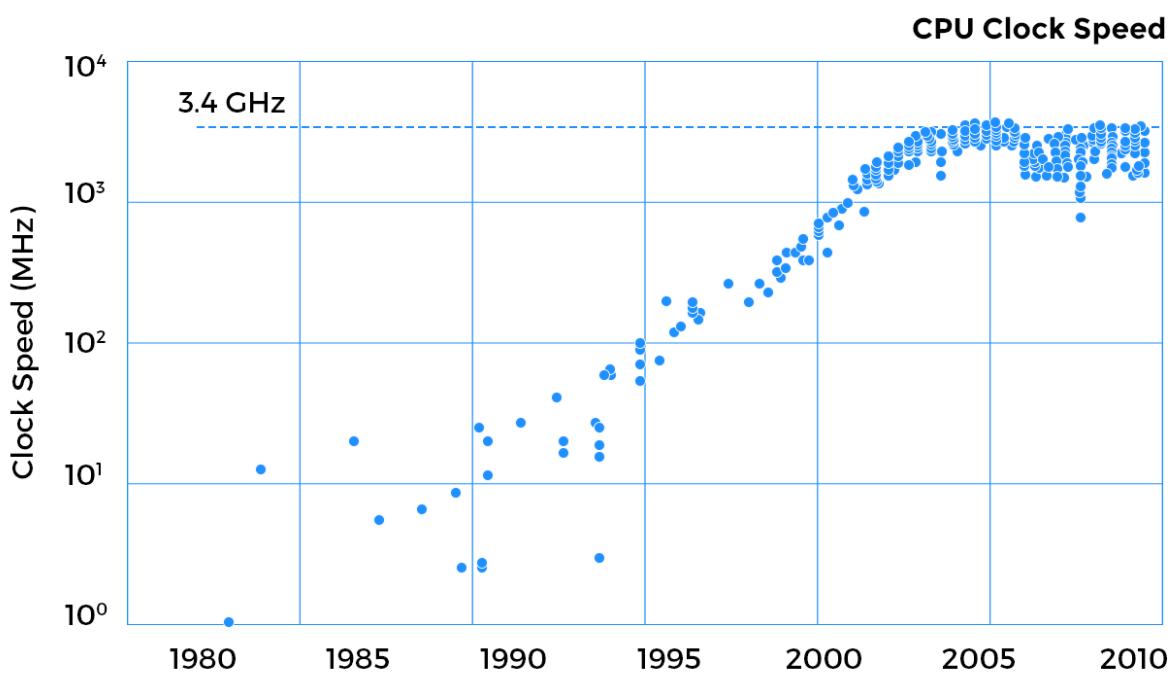
[Gordon Moore](#), co-founder of Intel, observed the number of transistors that can be packed into a given unit of space doubles about every two years and in turn the processing power of computers doubles and the cost halves. Moore's law is more of an observation than a law grounded in formal scientific 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

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.

# Thready Safety & Synchronized

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

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

## Thread Safe

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

## Synchronized

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

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

Note carefully:

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

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

```
class Employee {  
  
    // shared variable  
    private String name;  
  
    // method is synchronize on 'this' object  
    public synchronized void setName(String name) {  
        this.name = name;  
    }  
  
    // also synchronized on the same object  
    public synchronized void resetName() {  
  
        this.name = "";  
    }  
  
    // equivalent of adding synchronized in method  
    // definition  
    public String getName() {  
        synchronized (this) {  
            return this.name;  
        }  
    }  
}
```

As an example look at the employee class above. All the three methods are synchronized on the "**this**" object. If we created an object and three different threads attempted to execute each method of the object, only

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

```
class Employee {  
  
    // shared variable  
    private String name;  
    private Object lock = new Object();  
  
    // method is synchronize on 'this' object  
    public synchronized void setName(String name) {  
        this.name = name;  
    }  
  
    // also synchronized on the same object  
    public synchronized void resetName() {  
  
        this.name = "";  
    }  
  
    // equivalent of adding synchronized in method  
    // definition  
    public String getName() {  
        // Using a different object to synchronize on  
        synchronized (lock) {  
            return this.name;  
        }  
    }  
}
```

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

Note with the use of the synchronized keyword, **Java forces you to**

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

A classic newbie mistake is to synchronize on an object and then somewhere in the code reassign the object. As an example, look at the code below. We synchronize on a Boolean object in the first thread but sleep before we call `wait()` on the object. While the first thread is asleep, the second thread goes on to change the `flag`'s value. When the first thread wakes up and attempts to invoke `wait()`, it is met with a **IllegalMonitorStateException** 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

## 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

## 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 InterruptedException.

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 InterruptedException 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());  
                }  
            }  
        });  
        sleepyThread.start();  
        Thread.sleep(1000);  
        sleepyThread.interrupt();  
    }  
}
```

```
System.out.println("The interrupt flag is cleared : " + Thread.interrupted());
Thread.currentThread().interrupt();
System.out.println("Oh someone woke me up ! ");
System.out.println("The interrupt flag is set now : " + Thread.currentThread().isInterrupted());
}

});

sleepyThread.start();

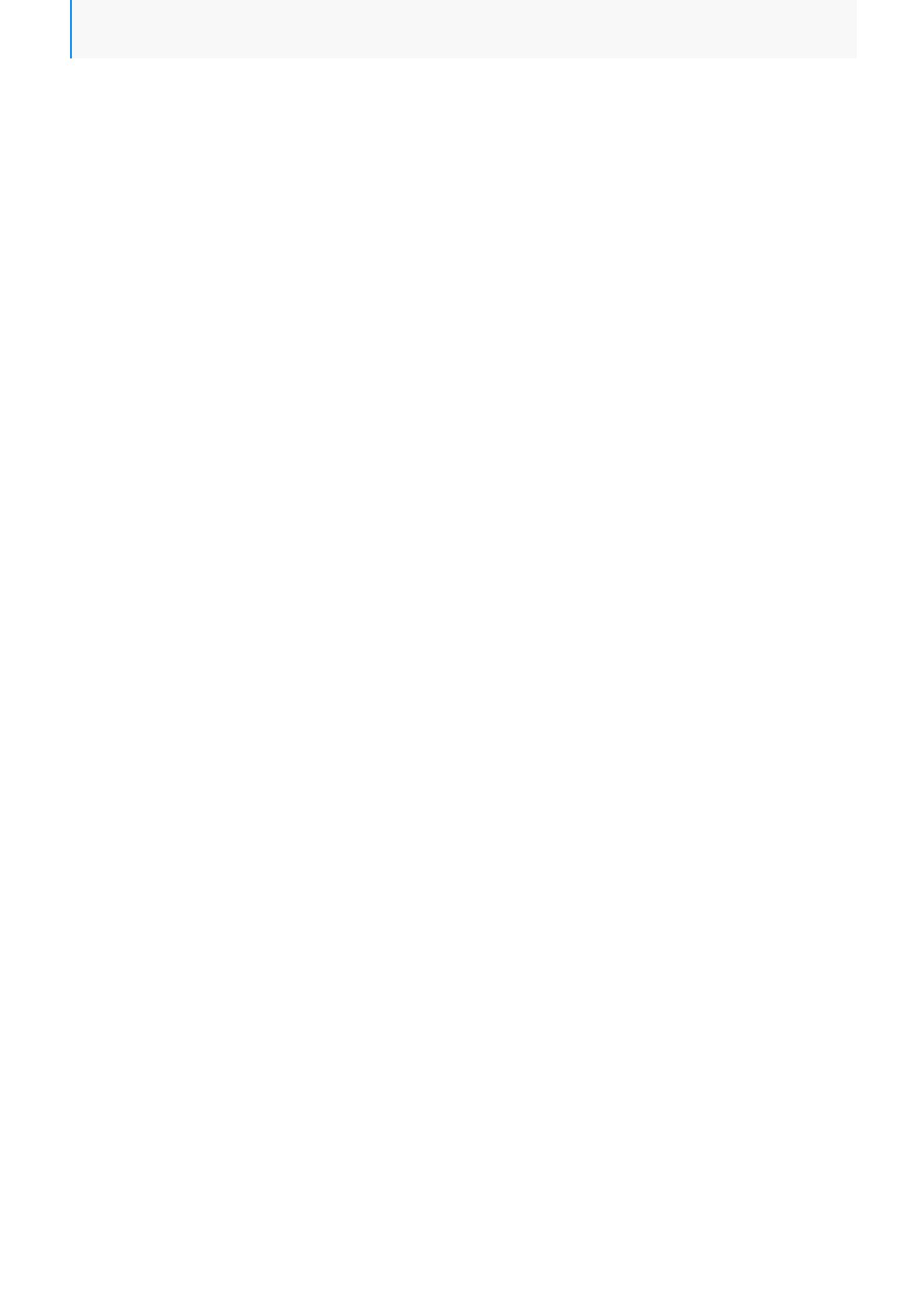
System.out.println("About to wake up the sleepy thread ...");
sleepyThread.interrupt();
System.out.println("Woke up sleepy thread ...");

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

## Volatile

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

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

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

Volatile comes into play because of 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

## Reentrant Lock

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

## Condition Variable

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

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

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

## java.util.concurrent

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

# 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.");
}
}

```



Missed Signal Example

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

```

import java.util.concurrent.Semaphore;

class Demonstration {

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

```



```
        }

class FixedMissedSignalExample {

    public static void example() throws InterruptedException {

        final Semaphore semaphore = new Semaphore(1);

        Thread signaller = new Thread(new Runnable() {

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

        Thread waiter = new Thread(new Runnable() {

            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.");
    }
}
```



Fixed Missed Signal

# Semaphore in Java

## Semaphore

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

```
import java.util.concurrent.Semaphore;  
  
class Demonstration {  
  
    public static void main(String args[]) throws InterruptedException {  
        IncorrectSemaphoreExample.example();  
    }  
}  
  
class IncorrectSemaphoreExample {  
  
    public static void example() throws InterruptedException {  
  
        final Semaphore semaphore = new Semaphore(1);  
  
        Thread badThread = new Thread(new Runnable() {  
  
            public void run() {  
  
                while (true) {  
  
                    try {  
                        semaphore.acquire();  
                    } catch (InterruptedException ie) {  
                        // handle thread interruption  
                    }  
  
                    // Thread was meant to run forever but runs into an  
                    // 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");
}
}

```



### Incorrect Use of Semaphore

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

```

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 semahore.");
                        semaphore.release();
                    }
                } catch (InterruptedException ie) {
                    // handle thread interruption
                }
            }
        });
    });

    badThread.start();

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

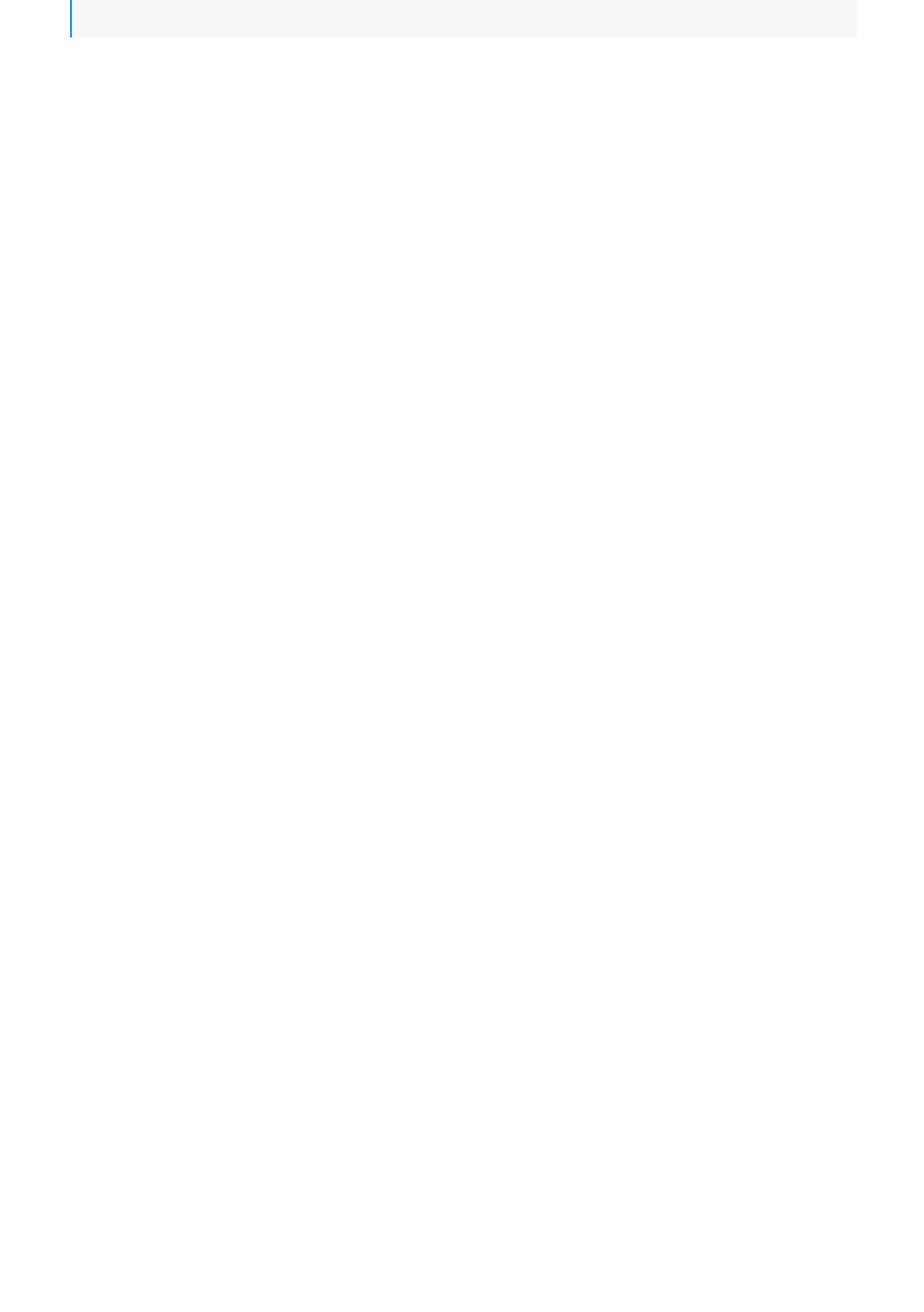
    final Thread goodThread = new Thread(new Runnable() {
        public void run() {
            System.out.println("Good thread patiently waiting to be signalled.");
            try {
                semaphore.acquire();
            } catch (InterruptedException ie) {
                // handle thread interruption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

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

```



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



# Spurious Wakeups

## Spurious Wakeups

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

```
* A thread can also wake up without being notified, interrupted, or
* timing out, a so-called <i>spurious wakeup</i>. While this will ra
rely
* occur in practice, applications must guard against it by testing fo
r
* the condition that should have caused the thread to be awakened and
* continuing to wait if the condition is not satisfied. In other wor
ds,
* waits should always occur in loops, like this one:
*
*     synchronized (obj) {
*         while (condition does not hold)
*             obj.wait(timeout);
*             ... // Perform action appropriate to condition
*     }
*
```



# Miscellaneous Topics

## Lock Fairness

We'll briefly touch on the topic of fairness in locks since it's 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.

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

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);  
    }  
}
```

```
}
```



### Effect of reordering on program output

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;
```

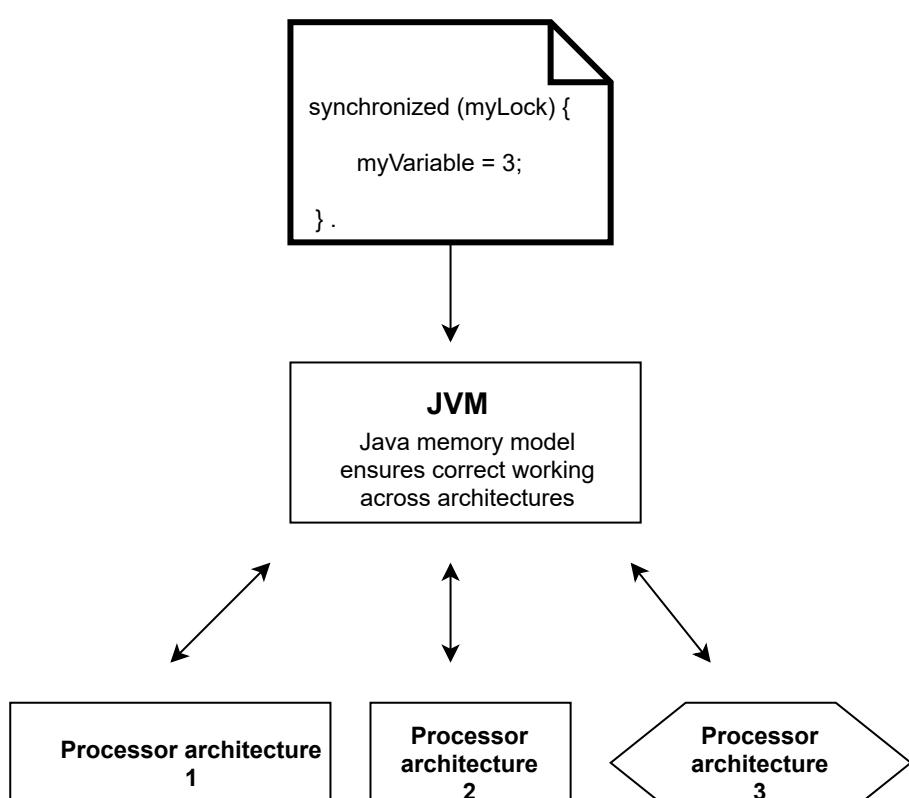
The two statements don't have a dependence on each other in the sense that they are working off of completely different variables. For performance reasons, the compiler may decide to switch their ordering. Other forces are also at play, for instance, the value of one of the variables may get flushed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

developers on guard and fear to bugs which are very hard to reproduce in production.

## Platform

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

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.

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 blo
ck#2
        synchronized (lock1) {
            x *= x;
            y *= y;
        }

        // BLOCK#4
        // Since this block is also not dependent on block#3, it ca
n be
        // placed before block#3 or block#2. But it can't be placed b
efore
        // 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.

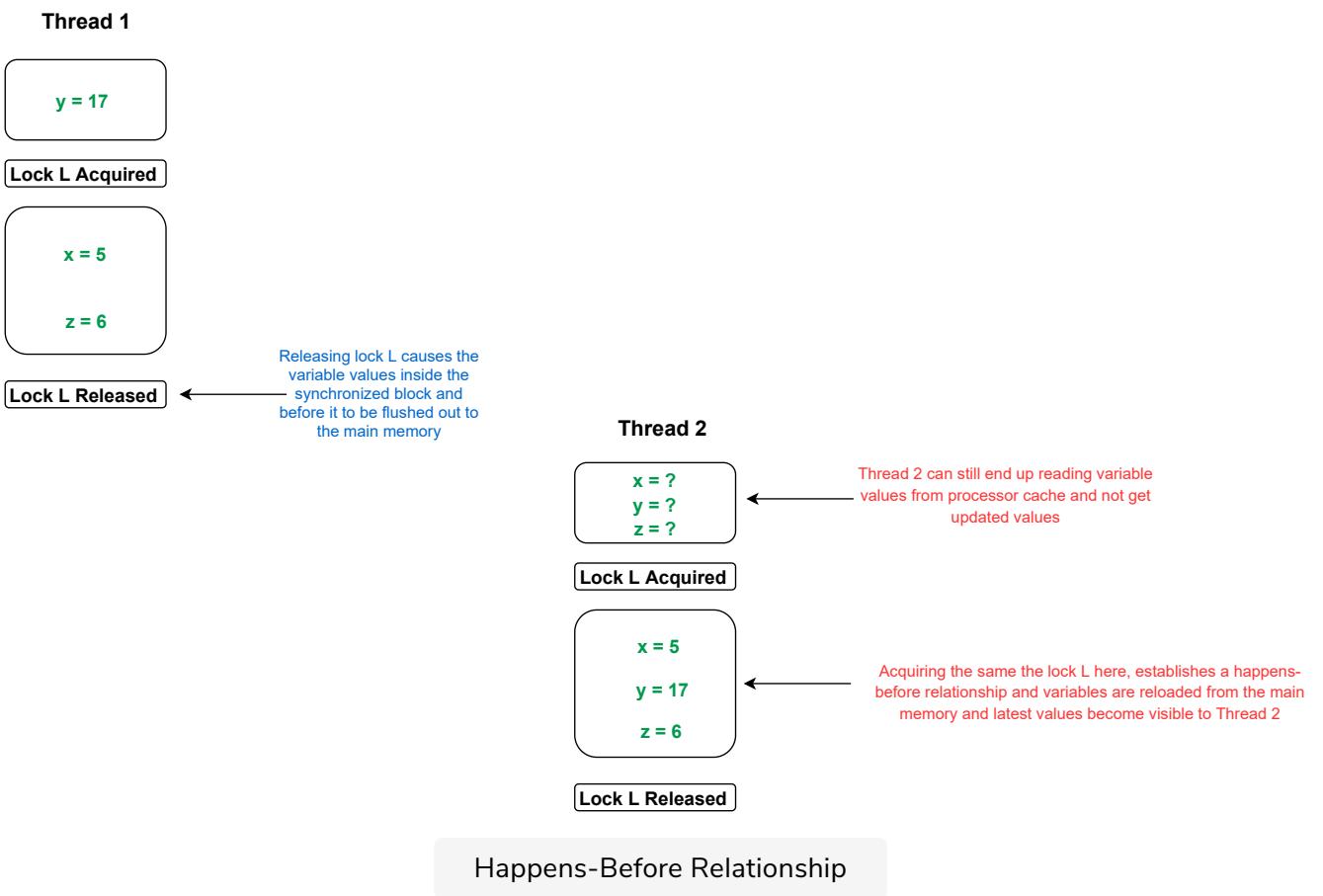
- Each action in a thread *happens-before* every action in that thread that comes later in the program's order. However, for a single-threaded program, instructions can be reordered but the semantics of the program order is still preserved.
- An unlock on a monitor *happens-before* every subsequent lock 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

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

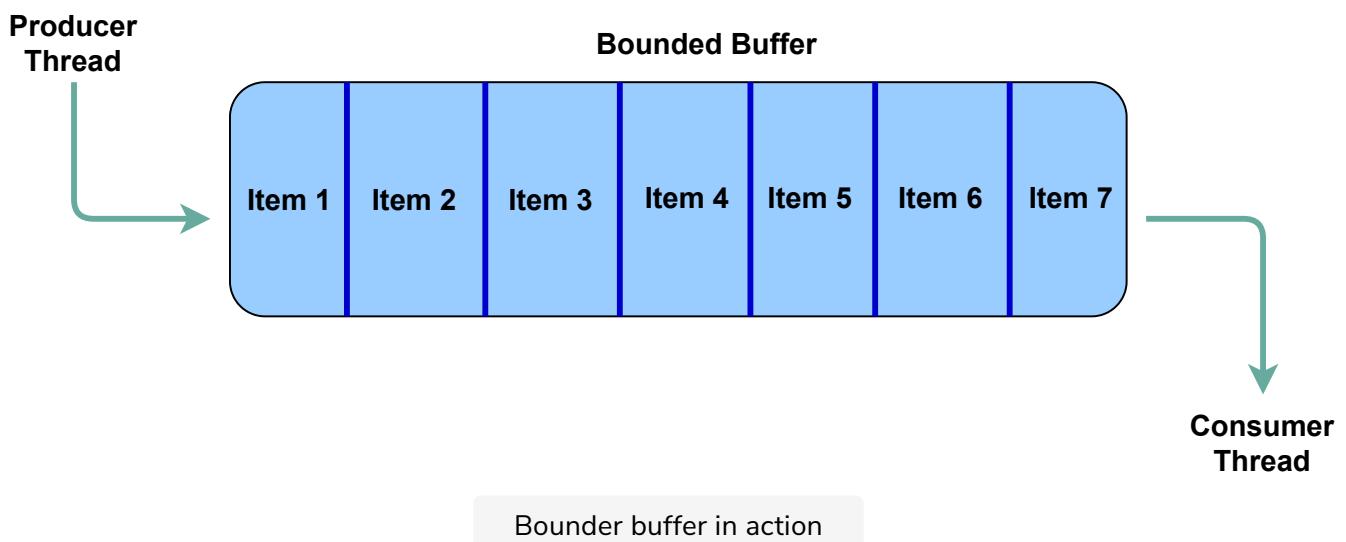


# Blocking Queue | Bounded Buffer | Consumer Producer

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.

## Problem

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

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 have to handle this case as well. We can do this by using a **linked list** instead of an array.

need to reset the tail of the queue back to zero. Notice that since we only

proceed to enqueue an item when `size < 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) {
    }
}}
```

```
        }
    } 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 enqueueer 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.

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 of queue < 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 a lock.

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



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

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

        try {
            Thread.sleep(1000);
        } catch (InterruptedException ie) {
        }

        q.shutdown();
        producer1.join();
        producer2.join();
        producer3.join();
    }
}
```

```
        System.out.println("Producer thread 3 enqueued " + i);
        i++;
    }
} 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);
}
```

```
}
```



## 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

FaultyBlockingQueueWithMutex

```
class Demonstration {

    static final FaultyBlockingQueueWithMutex<Integer> q = new FaultyBlockingQueueWithMutex<

    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);
}
}
```



## ... continued

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

### Using Semaphores for Producer-Consumer Problem

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();  
    array[tail] = item;  
    size++;  
    tail++;  
    semConsumer.release();  
    semLock.release();  
}
```

```

    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.



## BlockingQueueWithSemaphore

```
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();  
}  
}
```



# Rate Limiting Using Token Bucket Filter

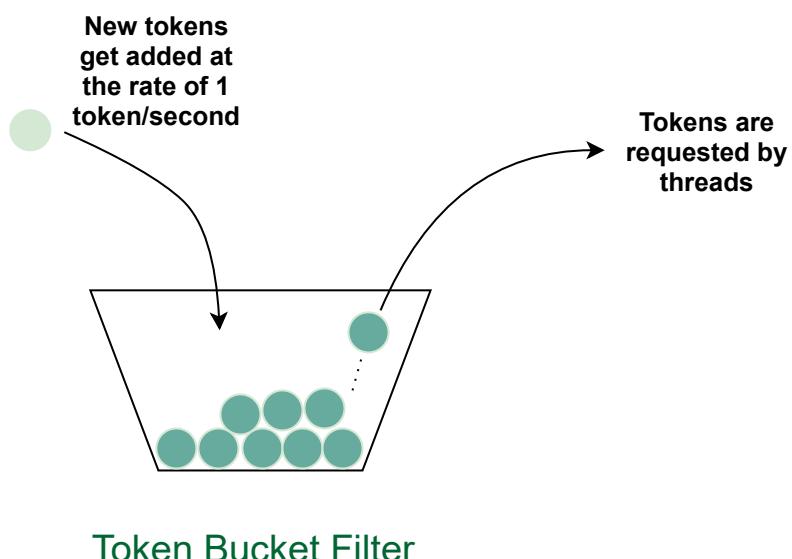
Implementing rate limiting using a naive token bucket filter algorithm.

## Problem

*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. 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 at " + (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

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 " + (S
    }

    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");
            }
        }
    });
    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 " + (S
}

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 Exercise

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.

### 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();

        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.curren
    }
}

```



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



TokenBucketFilter.java

TokenBucketFilterFactory.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();

        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();
        }

    }
}
```



# 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

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-

Concurrent 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() + executeAfte
r * 1000;
        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 Compa
rator<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() + executeAfte
r * 1000;
            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 captured 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);
    }
}

```

```
+ " : message:" + cb.message);

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

#### Executor thread's workflow

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.

```

        break;
    }

    CallBack cb = q.poll();
    System.out.println(
        "Executed at " + System.currentTimeMillis()/1000 + " required at " + cb.e
        + ": 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().get
                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();
            }
            CallBack cb = q.poll();
            cb.execute();
        }
    }
}
```

```

        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.e
        + ": 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() {

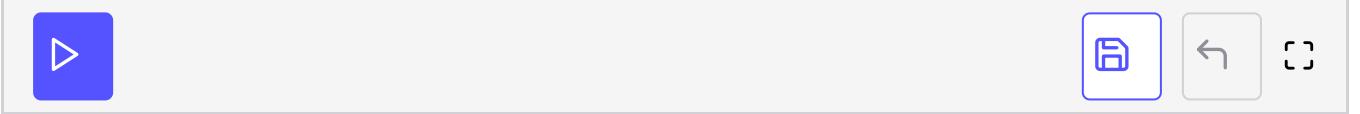
```

```
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

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

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.  
6.     public CountingSemaphore(int count) {  
7.         this.maxCount = count;  
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 {  
                    cs.acquire();  
                    System.out.println("t1 acquired");  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        });  
  
        Thread t2 = new Thread(new Runnable() {  
            @Override  
            public void run() {  
                try {  
                    cs.release();  
                    System.out.println("t2 released");  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        });  
  
        t1.start();  
        t2.start();  
    }  
}
```

```

        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

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 in it in the critical section

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 {  
    }  
  
    public synchronized void releaseReadLock() {  
    }  
  
    public synchronized void acquireWriteLock() throws InterruptedException {  
        if (isWriteLocked) {  
            throw new InterruptedException("Writer is already writing");  
        }  
        isWriteLocked = true;  
    }  
  
    public synchronized void releaseWriteLock() {  
        if (!isWriteLocked) {  
            throw new InterruptedException("Writer is not writing");  
        }  
        isWriteLocked = false;  
    }  
}
```

```
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 {
    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

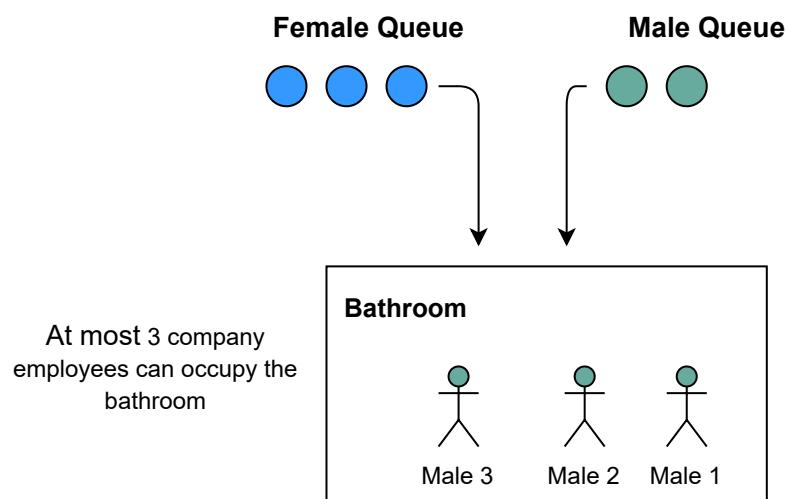
A synchronization practice problem requiring us to synchronize the usage of a single bathroom by both the genders.

## Problem

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

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 employ  
ees 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 associat  
ed  
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

The complete code appears below:

```
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 = "
            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 `empsIBathroom` 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

- Try writing a solution in which there's no possibility of starvation for threads of either gender.



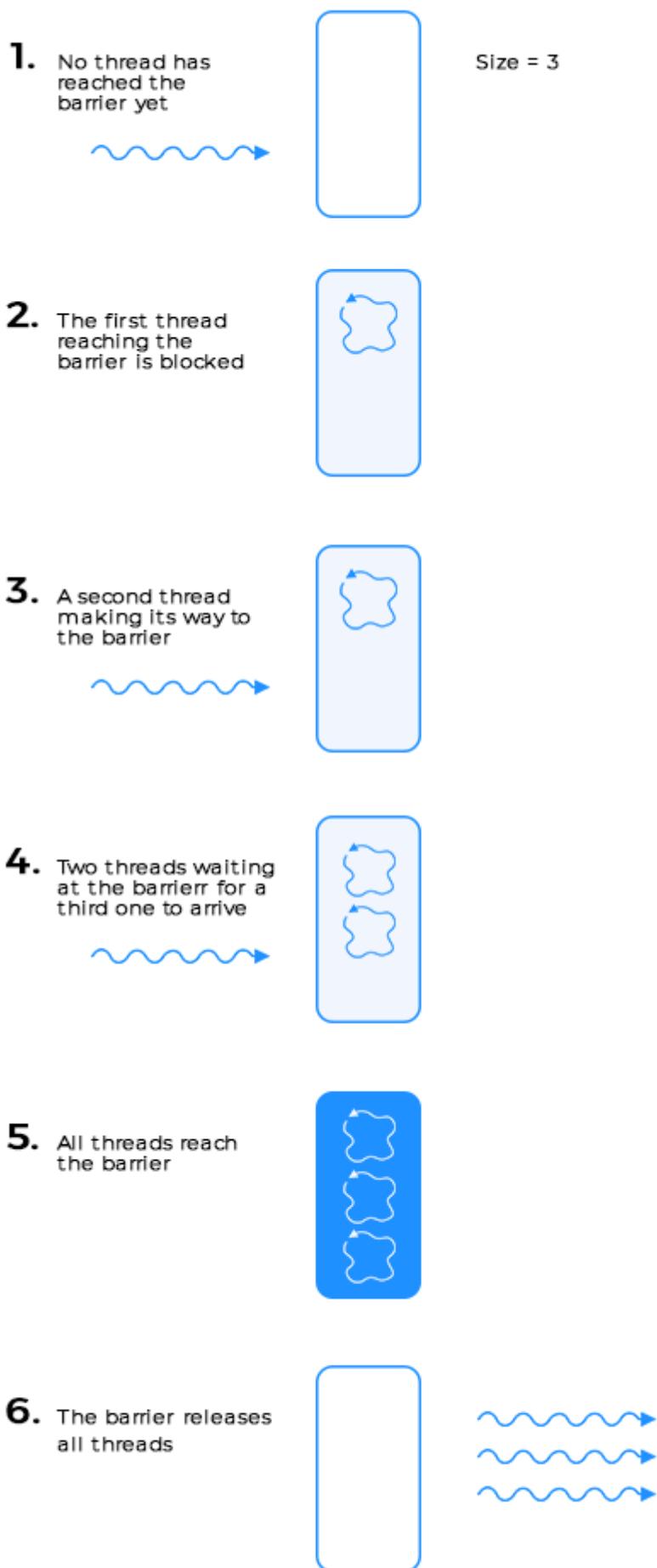
# Implementing a Barrier

This lesson discusses how a barrier can be implemented in Java.

## Problem

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



## 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 **nth** thread to arrive. This suggests we have the **n-1** threads execute the wait method and the **nth** 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.    }
```

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 threds.
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.}
```

```
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();  
    }  
}  
}
```



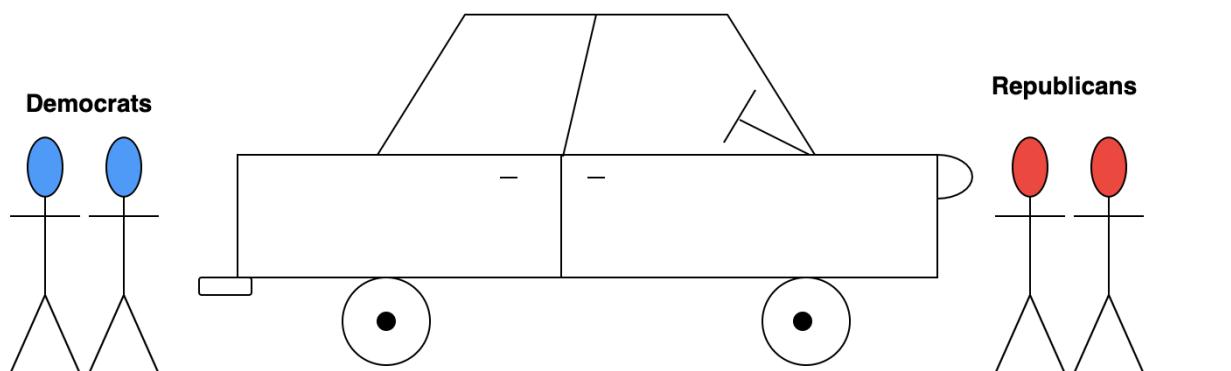
# Uber Ride Problem

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

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.



Uber Seating Problem

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 {
            // Seat 1 democrat & 1 republican
            demsWaiting.release(1);
            repubsWaiting.release(1);
            rideLeader = true;
            democrats -= 1;
            republicans -= 1;
        }

        barrier.await();
    }
}
```

```

} 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 one party. In this case, the barrier wait time is the same for both

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.

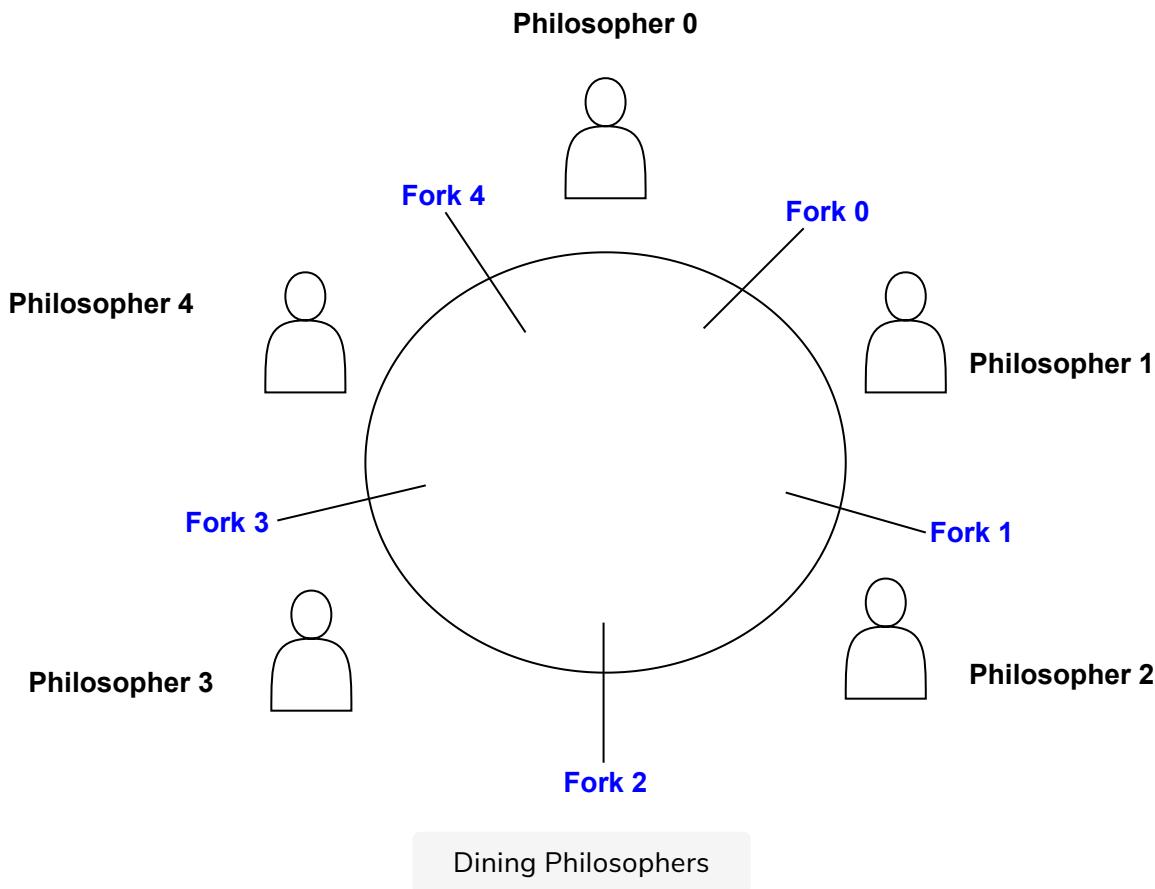
## Problem

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.

### Limits 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();
}
}

```

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();
}
}
```

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();
}
}
```



Dining Philosopher Solution

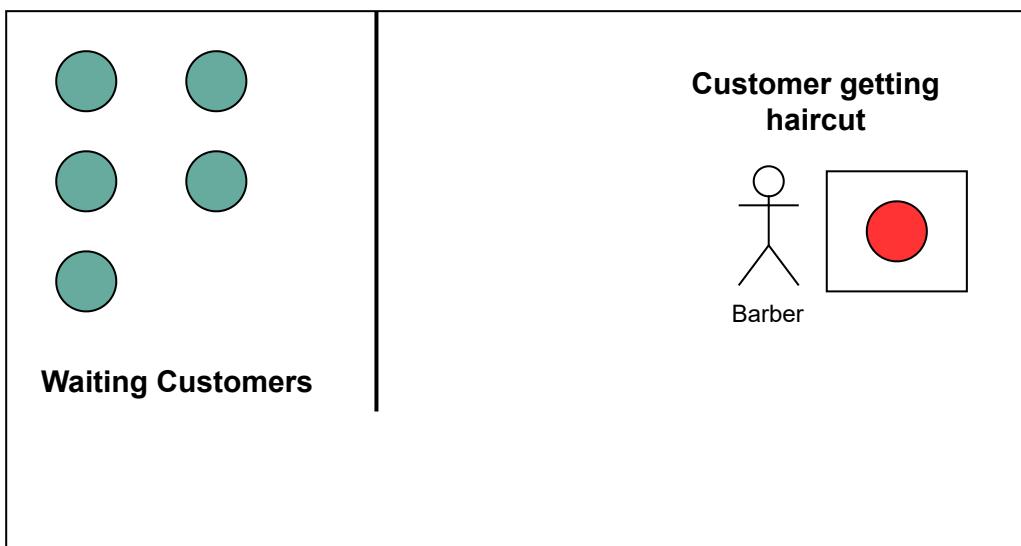
# Barber Shop

This lesson visits the synchronization issues when programmatically modeling a hypothetical barber shop and how they are solved using Java's concurrency primitives.

## Problem

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 its 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 in, all chairs occupied");
    }
    waitingCustomers++;
}
```

```
System.out.println("Customer walks out, all chairs occupied");
ed");
    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 vacant
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) {

            }
        }
    });
    set.add(barberThread);
    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 {
```

```

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

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 SupermanNaiveButCorrect();  
  
    // 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 initialize 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

The complete code appears below:

```
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.

```
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 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() {  
    }  
  
    synchronized public static Superman getInstance() {  
        if (superman == null) {  
            superman = new Superman();  
        }  
  
        return 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 ...");
    }
}
```



- The next evolution is the double checked locking that we have already discussed in the previous lesson.
- Another implementation of the singleton pattern is the **holder** or **Bill Pugh's** singleton. The idea is to create a private nested static class that holds the static instance. The nested class **Helper** isn't loaded when the outer class **Superman** is loaded. The inner static class **Helper** is loaded only when the method **getInstance()** is invoked. This saves us from eagerly initializing the singleton instance.

```
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**

$$T(n) = \text{Cost to divide into 2 unsorted arrays} + 2 * T\left(\frac{n}{2}\right) + C$$

cost to merge 2 sorted arrays when  $n > 1$

$$T(n) = O(1) \text{ when } n = 1$$

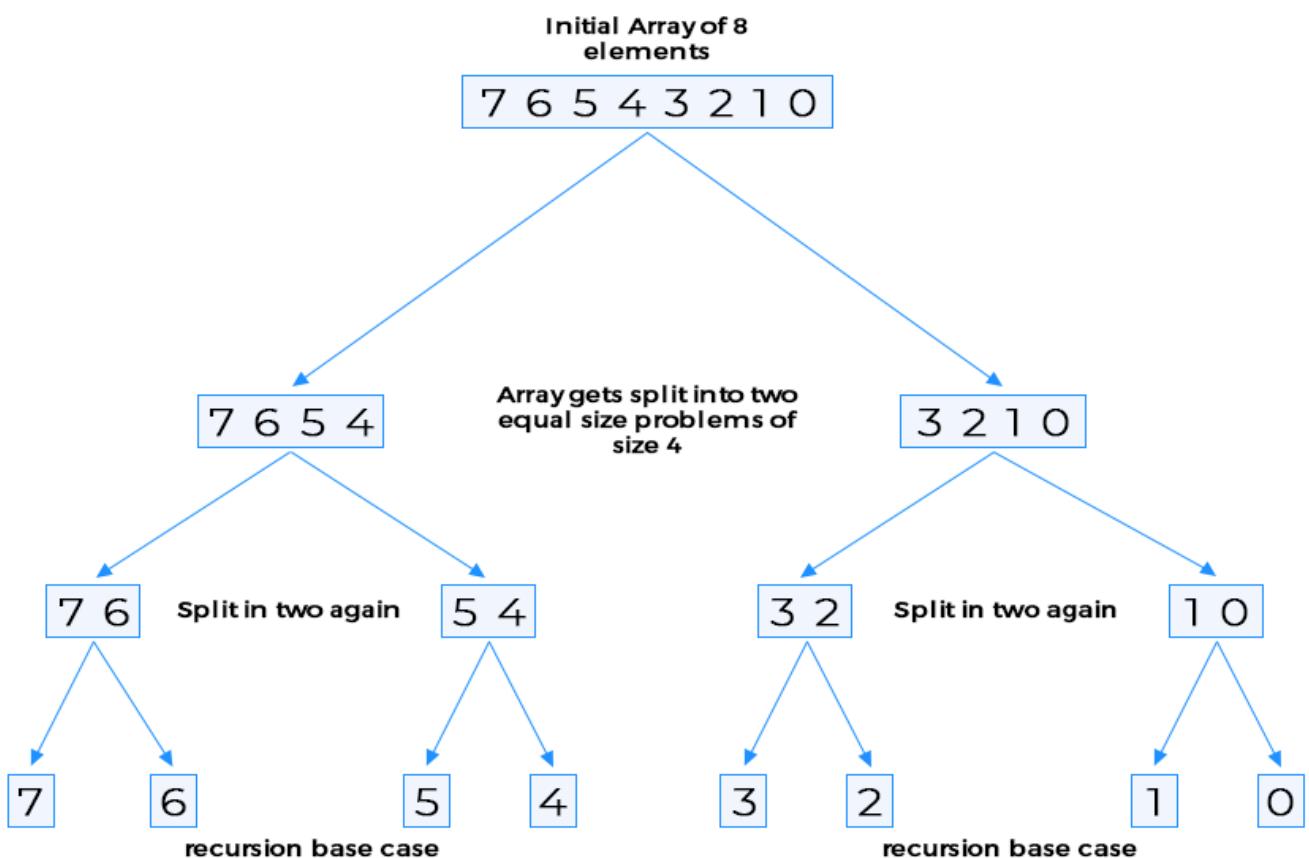
Remember the *solution* to the recurrence equation will be the *running time* of the algorithm on an input of size  $n$ . Without getting into the details of how we'll solve the recurrence equation, the running time of merge sort is

$$O(n \lg n)$$

where  $n$  is the size of the input array.

### Merge Sort Recursion Tree

Below is a pictorial representation of how the merge sort algorithm works



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.         printArray(input,"Before: ");  
8.         mergeSort(0, input.length-1, input);  
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.    }  
}
```

```

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++;
}
}
}

```

```
}
```



### Multithreaded Merge Sort

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

*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() {
```

```
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();
        }
    }
}
```

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.

# 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.**

You can find the solutions to the problems discussed in this course at the following github repo:

## **Github Repo**

Every great product is a result of team-effort and so is this course.

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\*\*](#)

# Ordered Printing

This problem is about imposing an order on thread execution.

## Problem

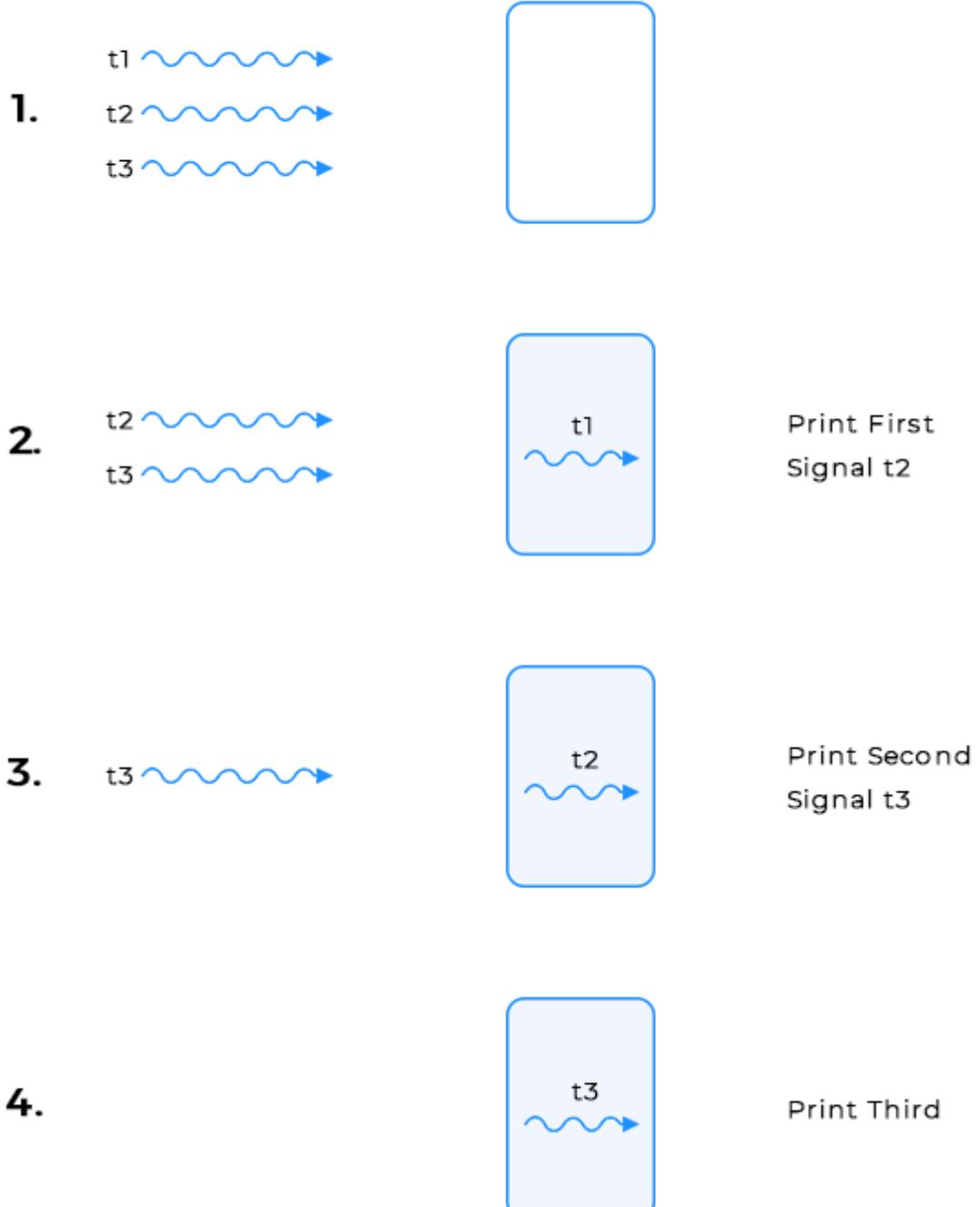
Suppose there are three threads t1, t2 and t3. t1 prints **First**, t2 prints **Second** and t3 prints **Third**. The code for the class is as follows:

```
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.obj = obj;
        this.method = 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 2

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, decreasing the count by 1. Once this count reaches zero, all the threads waiting for the count to reach zero are woken up.

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 {
```

```
    public void printSecond() throws InterruptedException {
        //wait if "Second" has not been printed yet
        latch2.await();
        System.out.println("Second");
    }
}
```

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

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");  
        }  
    }  
}
```

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:

## Time



## 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 **synchronized(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) {
                if (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.

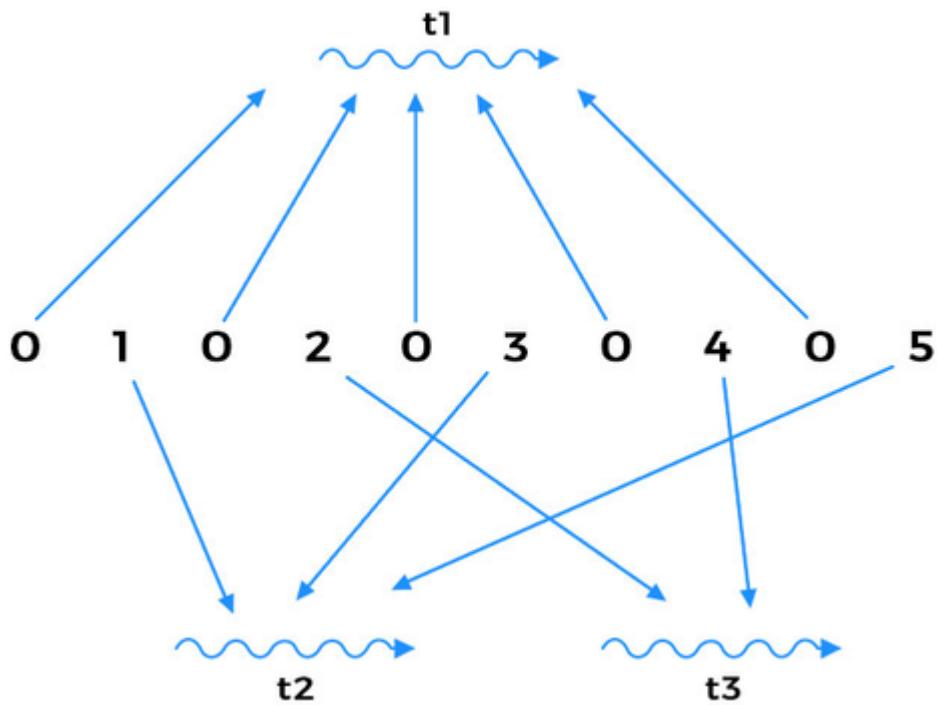
## Problem

Suppose we are given a number  $n$  based on which a program creates the series  $010203\dots 0n$ . 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 `PrintZero()`, `PrintOdd()` and `PrintEven()` are executed in an order.

The workflow of the program is shown below:



Workflow

## 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.

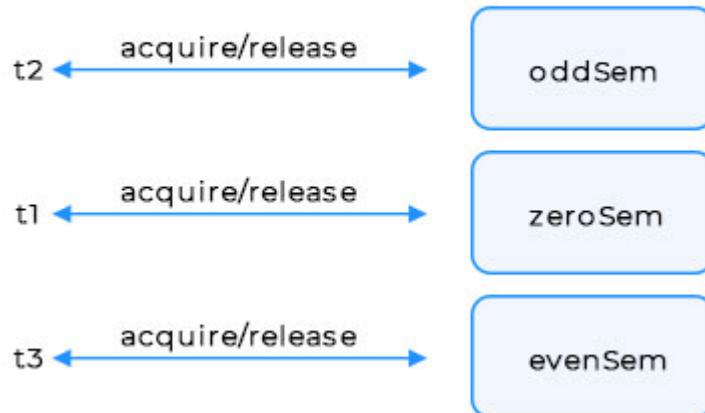
The last method of the class **PrintEven()** is shown below:

```
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();

    }
}
```



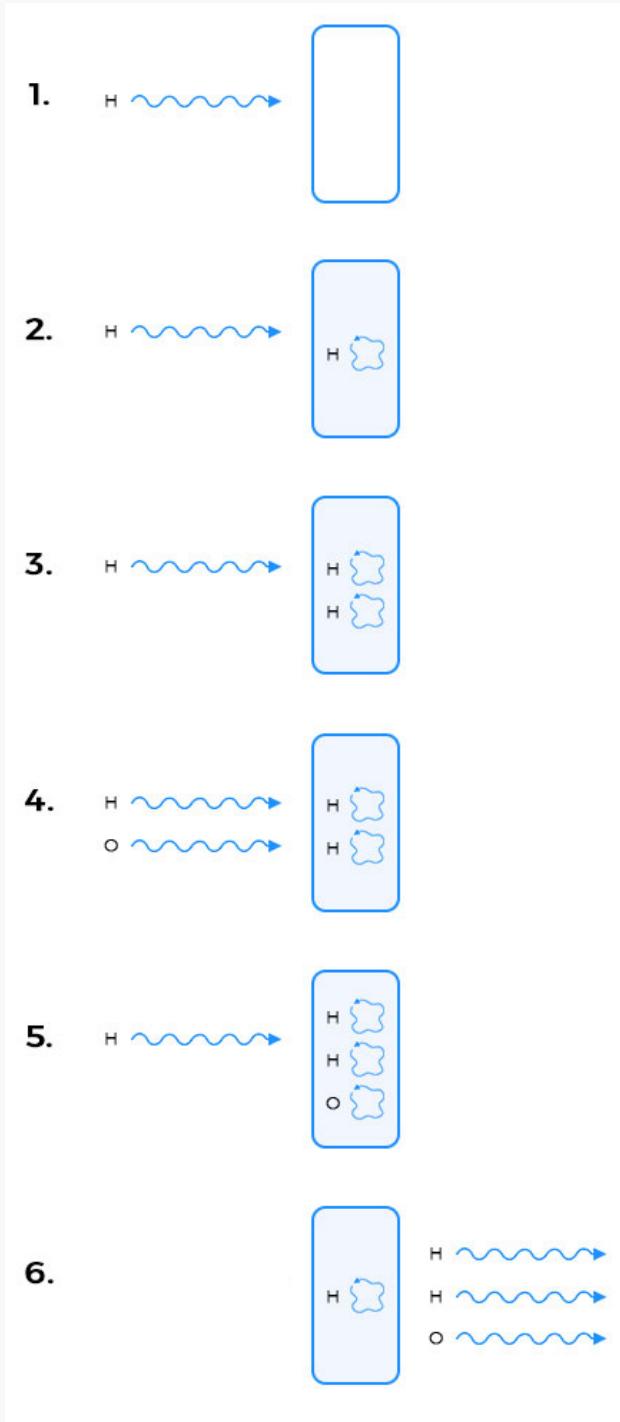
We were told to use three threads in the problem statement but the solution can be achieved using two threads as well. Since zero is printed before every number, we do not need to dedicate a special thread for it. We can simply print a zero before printing every odd or even number.

# Build a Molecule

This problem simulated the creation of water molecule by grouping three threads representing Hydrogen and Oxygen atoms.

## Problem

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:

```
class 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++;

        // 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 H<sub>2</sub>O 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) {
        }
    }
}
}

```

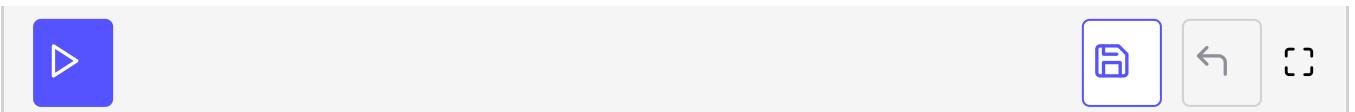
```
        }
    }
}

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();
    }
}
```



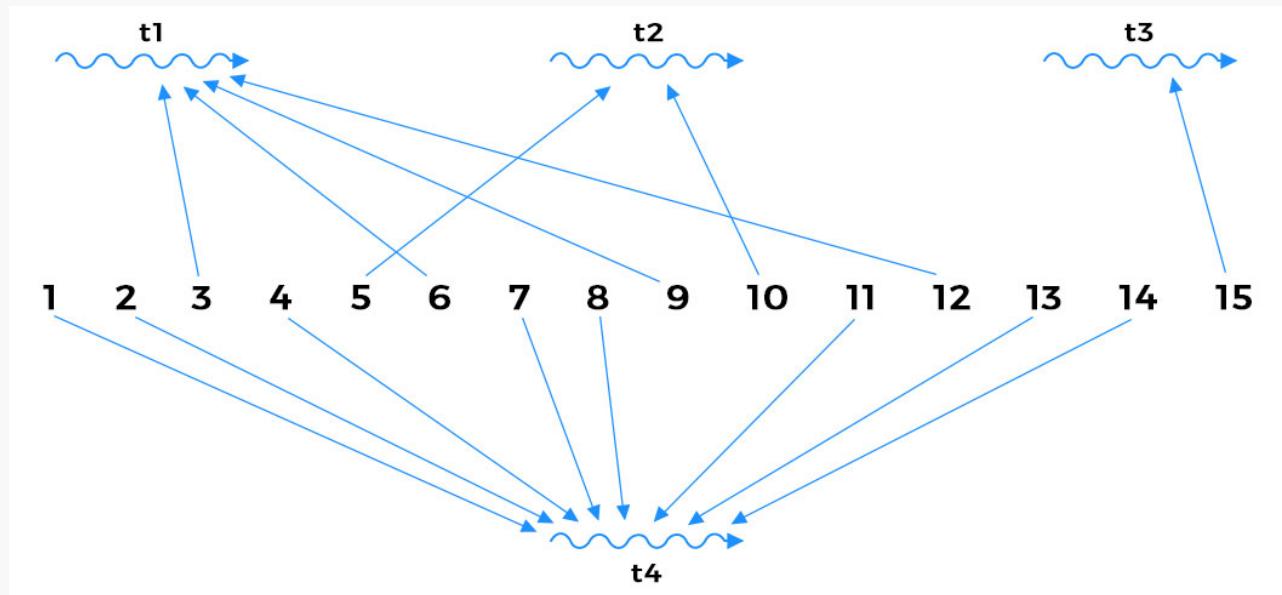
# Fizz Buzz Problem

This problem explores a multi-threaded solution to the very common Fizz Buzz programming task

## Problem

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 is that it checks if `num` is divisible by 5 and prints "buzz".

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 {
```

```

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) {
    }
}

}

}

public 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();
    }
}
```



## Next Steps

Moving from interview mindset to learning mindset.

In the Beyond the Interview section, we'll go into further depth and detail of various concurrency topics in Java. The focus of the section will be on learning concurrency for the long-term rather than a stop-gap fix to pass interview questions. Note this section will be an ongoing work with updates and new sections from time to time.

# 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 output from 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.

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());  
                Thread.sleep(1000 * 1000);  
            } catch (InterruptedException ie) {  
                System.out.println("innerThread interrupted at " + +System.currentTimeMillis());  
            }  
        }  
    }  
}
```



# 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 pre-configured 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 life-cycle:

- Running
- Shutting Down
- Terminated

As mentioned earlier, JVM can't exit unless all non-daemon threads 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

Contrasting Timer and 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);  
    }  
}
```



# 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();
    }
}
```

```
        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());
        }
    }
}
```

```
        System.out.println( "Something went wrong." + ee.getCause()),  
    }  
  
    return result;  
}  
  
}
```



### Callable throwing Exception

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("\nIs second task cancelled : " + f2.isCancelled());

return result;
}
}

```



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;
    }
});

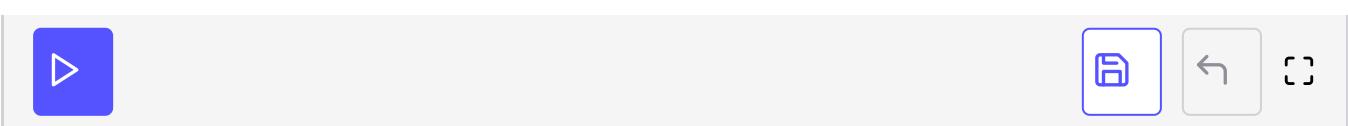
ExecutorService threadPool = Executors.newSingleThreadExecutor();
Future duplicateFuture = threadPool.submit(futureTask);

// 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.



```

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 don't terminate and are returned to the threadpool. So any threadlocal variables 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);  
    }  
}
```





# CountDownLatch

## CountDownLatch

`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

Worker.java

Master.java

```
public class Master extends Thread
{
    public Master(String name)
    {
        super(name);
    }

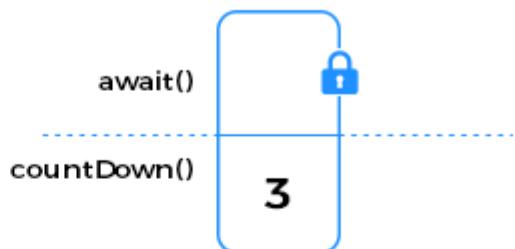
    @Override
    public void run()
    {
        System.out.println("Master executed "+Thread.currentThread().getName());
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
    }
}
```



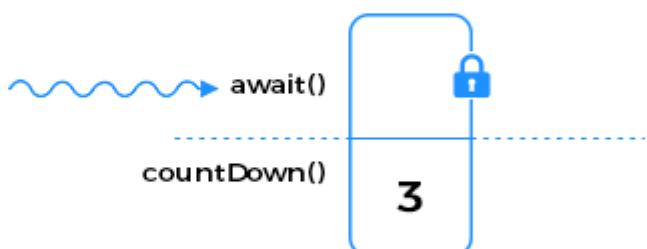
A pictorial representation appears below:

## CountDownLatch

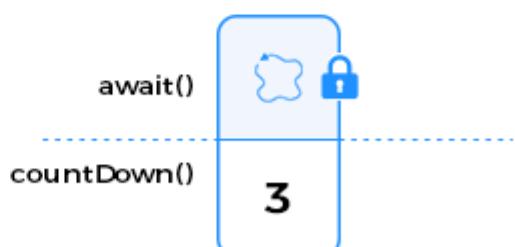
1. CountDownLatch initialized with a count of 3



2. A thread invokes await() on the CountDownLatch object



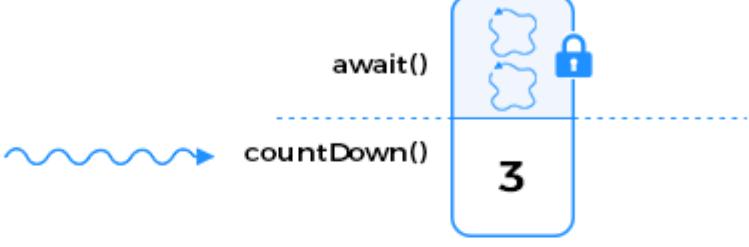
3. Thread is blocked till the count reaches 0



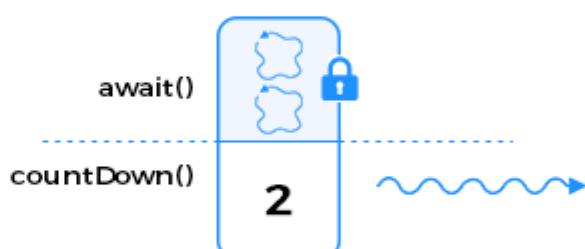
4. A second thread invokes await() and gets blocked



5. Two threads waiting for count to reach 0 and another thread invokes countDown()



6. Count is now 2



7. Another thread invokes



countDown()



countDown()

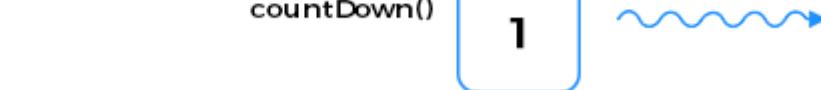
2

8. Count is decremented to 1

await()

countDown()

1



9. A third thread invokes countDown()



await()

countDown()

1

10. Count reaches 0

await()

countDown()

0



11. Latch opens and the two blocked threads are allowed to proceed.

await()

countDown()

0



# CyclicBarrier

## CyclicBarrier

**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.

**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 need
        //s 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

Task.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();
    }
}
```

▶

↶ ↻

A pictorial representation appears below:

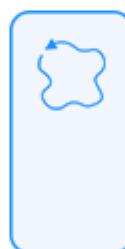
## Working of a Barrier

1. No thread has reached the barrier yet



Size = 3

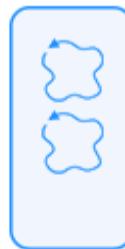
2. The first thread reaching the barrier is blocked



3. A second thread making its way to the barrier



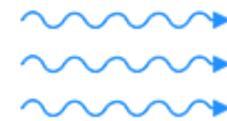
4. Two threads waiting at the barrier for a third one to arrive



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

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")  
    }  
}
```





# Quiz 1

## Question # 1

***What are some of the differences between a process and a thread?***

- A process can have many threads, whereas a thread can belong to only one process.
- A thread is lightweight than a process and uses less resources than a process.
- A thread has some state private to itself but threads of a process can share the resources allocated to the process including memory address space.

## Question # 2

***Given the below code, can you identify what the coder missed?***

```
void defectiveCode(final int n) throws ExecutionException, InterruptedException {  
  
    ExecutorService threadPool = Executors.newFixedThreadPool(5);  
  
    Callable<Void> sumTask = new Callable<Void>() {  
  
        public Void call() throws Exception {  
            System.out.println("Running");  
        }  
    };  
  
    Future<Void> result = threadPool.submit(sumTask);  
    result.get();  
}
```

```
        return null;
    }

};

threadPool.submit(sumTask);
f.get();
}
```

The above code forgets to **shutdown** the executor thread pool. The thread pool when instantiated would also create 5 worker threads. If we don't shutdown the executor when exiting the main method, then JVM would also not exit. It will keep waiting for the pool's worker threads to finish, since they aren't marked as daemon. As an example execute the below code snippet.

```
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

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?***

Q

Check Answers

#### 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");  
}
```

Q

Check Answers

### 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");  
}
```

Q

Check Answers

### 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?***

Q

[Check Answers](#)

**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?***

Q

**Check Answers**

# 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;  
    }  
}
```

[Check Answers](#)

### 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 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

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 incremented, 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();

    }
}
```



**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 in
stance 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 in
stance 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 {
        // 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();

    }
}
```



### 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 ?");  
        }  
    }  
}
```

Q

Check Answers

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");
```

Q

Check Answers

Show Explanation

Question # 4

***An instance method synchronizes on the instance object, do you know what object do static methods synchronize on?***

Show Explanation

# 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);  
    }  
}
```

Q



Show Explanation

## 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.

```
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;
}
```

Q

Check Answers

Show Explanation

# 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?***

Q

Check Answers

Show Explanation

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;
}
}
```

Q

Check Answers

Show Explanation

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++) {  
        System.out.println(counter.get());  
    }  
}
```

```
        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());
    }
```

Q

COMPLETED 0%

1 of 1



Show Explanation

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();
}
```

Q

COMPLETED 0%

1 of 1



Show Explanation

### Question # 3

***What would have been the output of the print statement from the previous question if we created a pool with 20 threads?***

Q

COMPLETED 0%

1 of 1



Show Explanation

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(0);
    }

    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?**

Q

COMPLETED 0%

1 of 1



Show Explanation

```
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`.