**Introduction**

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

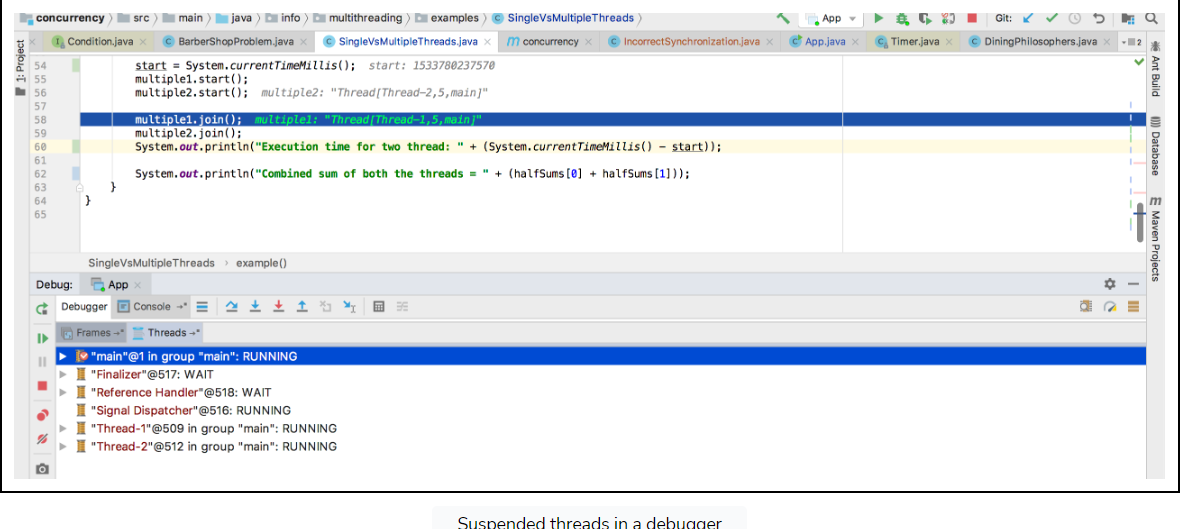
**We'll cover the following**

* [What good is concurrency?](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/introduction#What-good-is-concurrency)
* [Benefits of Threads](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/introduction#Benefits-of-Threads)
* [Performance Gains via Multi-Threading](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/introduction#Performance-Gains-via-Multi-Threading)
* [Problems with Threads](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/introduction#Problems-with-Threads)

**What good is concurrency?**

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

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



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.

**Problems with Threads**

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

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

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

**Program vs Process vs Thread**

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

**We'll cover the following**

* [Program](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/program-vs-process-vs-thread#Program)
* [Process](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/program-vs-process-vs-thread#Process)
* [Thread](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/program-vs-process-vs-thread#Thread)
* [Caveats](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/program-vs-process-vs-thread#Caveats)
* [Counter Program](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/program-vs-process-vs-thread#Counter-Program)
* [Thread unsafe class](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/program-vs-process-vs-thread#Thread-unsafe-class)

**Program**

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

**Process**

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

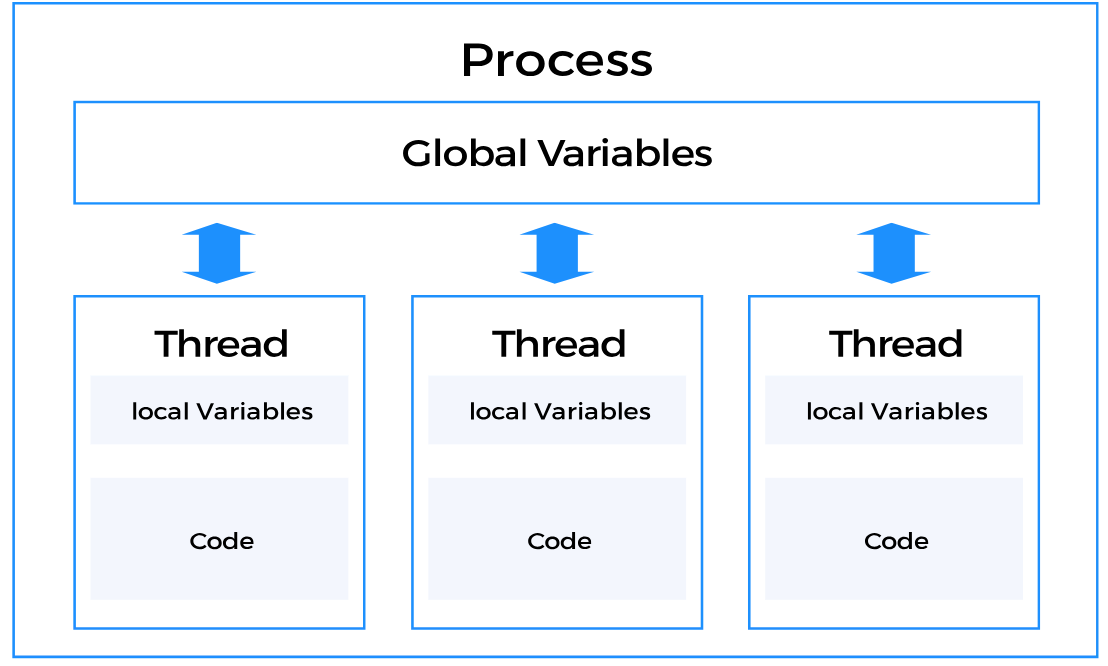
**Thread**

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

**Caveats**

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

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



**Counter Program**

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

Consider the below snippet of code

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

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

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

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

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

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

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

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

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

**Thread unsafe class**

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

import java.util.Random;

class DemoThreadUnsafe {

// We'll use this to randomly sleep our threads

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

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

// create object of unsafe counter

ThreadUnsafeCounter badCounter = new ThreadUnsafeCounter();

// setup thread1 to increment the badCounter 200 times

Thread thread1 = new Thread(new Runnable() {

@Override

public void run() {

for (int i = 0; i < 100; i++) {

badCounter.increment();

DemoThreadUnsafe.sleepRandomlyForLessThan10Secs();

}

}

});

// setup thread2 to decrement the badCounter 200 times

Thread thread2 = new Thread(new Runnable() {

@Override

public void run() {

for (int i = 0; i < 100; i++) {

badCounter.decrement();

DemoThreadUnsafe.sleepRandomlyForLessThan10Secs();

}

}

});

// run both threads

thread1.start();

thread2.start();

// wait for t1 and t2 to complete.

thread1.join();

thread2.join();

// print final value of counter

badCounter.printFinalCounterValue();

}

public static void sleepRandomlyForLessThan10Secs() {

try {

Thread.sleep(random.nextInt(10));

} catch (InterruptedException ie) {

}

}

}

class ThreadUnsafeCounter {

int count = 0;

public void increment() {

count++;

}

public void decrement() {

count--;

}

void printFinalCounterValue() {

System.out.println("counter is: " + count);

}

}

**Concurrency vs Parallelism**

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

**We'll cover the following**

* [Introduction](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/concurrency-vs-parallelism#Introduction)
* [Serial Execution](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/concurrency-vs-parallelism#Serial-Execution)
* [Concurrency](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/concurrency-vs-parallelism#Concurrency)
* [Parallelism](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/concurrency-vs-parallelism#Parallelism)
* [Concurrency vs Parallelism](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/concurrency-vs-parallelism#Concurrency-vs-Parallelism)

**Introduction**

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

**Serial Execution**

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

**Concurrency**

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

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

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

Going back to our circus analogy, a concurrent juggler is one who can juggle several balls at the same time. However, at any one point in time, he can only have a single ball in his hand while the rest are in flight. Each ball gets a time slice during which it lands in the juggler's hand and then is thrown back up. A concurrent system is in a similar sense *juggling* several processes at the same time.

**Parallelism**

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

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

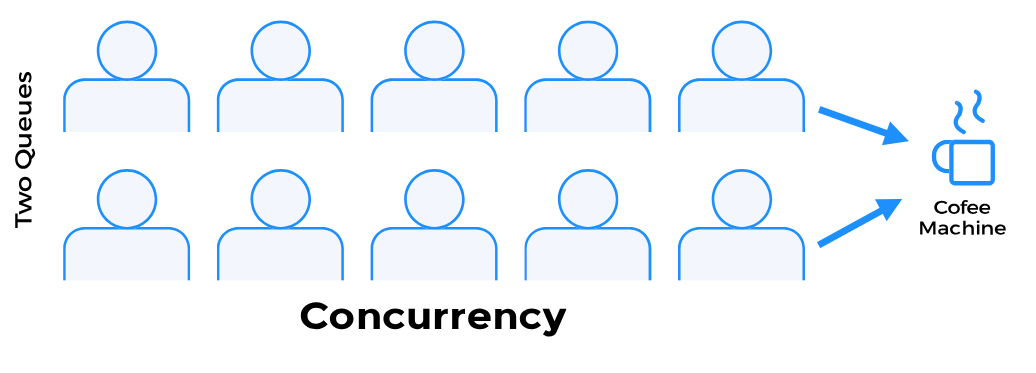
Revisiting our juggler analogy, a parallel system would map to at least two or more jugglers juggling one or more balls. In the case of an operating system, if it runs on a machine with say four CPUs then the operating system can execute four tasks at the same time, making execution parallel. Either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

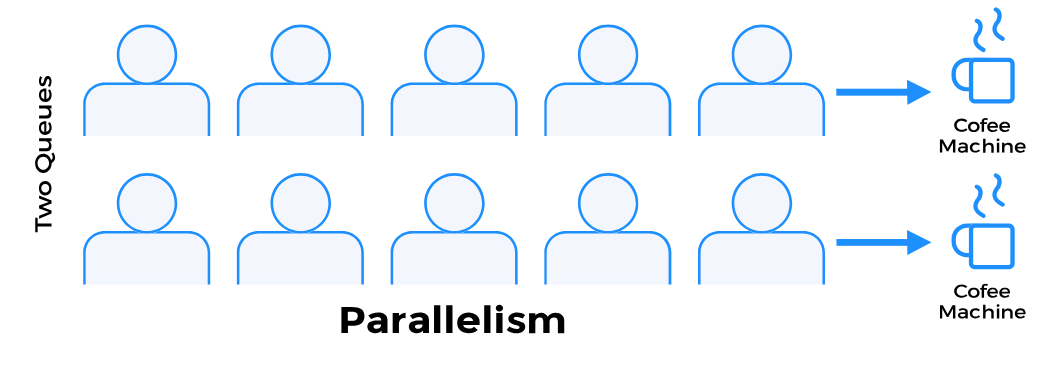
**Concurrency vs Parallelism**

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

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

We end the lesson with an analogy, frequently quoted in online literature, of customers waiting in two queues to buy coffee. Single-processor concurrency is akin to alternatively serving customers from the two queues but with a single coffee machine, while parallelism is similar to serving each customer queue with a dedicated coffee machine.





**Cooperative Multitasking vs Preemptive Multitasking**

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

**We'll cover the following**

* [Introduction](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/cooperative-multitasking-vs-preemptive-multitasking#Introduction)
* [Preemptive Multitasking](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/cooperative-multitasking-vs-preemptive-multitasking#Preemptive-Multitasking)
* [Cooperative Multitasking](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/cooperative-multitasking-vs-preemptive-multitasking#Cooperative-Multitasking)
* [Cooperative vs Preemptive](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/cooperative-multitasking-vs-preemptive-multitasking#Cooperative-vs-Preemptive)

**Introduction**

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

* Preemptive Multitasking
* Cooperative Multitasking

**Preemptive Multitasking**

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

**Cooperative Multitasking**

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

**Cooperative vs Preemptive**

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

**Synchronous vs Asynchronous**

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

**We'll cover the following**

* [Synchronous](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/synchronous-vs-asynchronous#Synchronous)
* [Asynchronous](https://www.educative.io/courses/java-multithreading-for-senior-engineering-interviews/synchronous-vs-asynchronous#Asynchronous)

**Synchronous**

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

**Asynchronous**

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

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

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