Concurrency and Multithreading

Created	@February 12, 2022 10:42 AM
Reviewed	☐

Table of contents

Concurrency and Multithreading

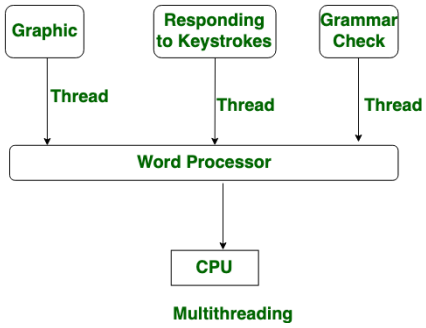
▼ Introduction

▼ Multitasking

One CPU can run multiple prorgams at a time by switching between executing **one program** at a time for a little time, and then **switching to next program**. Combination of **CPU and the operating system** decide which application should run for how much time and in what sequence.

▼ Multithreading

It is a model of program execution that allows for **multiple threads to be created within a process**, executing independently but concurrently sharing process resources. Depending on the hardware, threads can run fully parallel if they are distributed to their own CPU core.

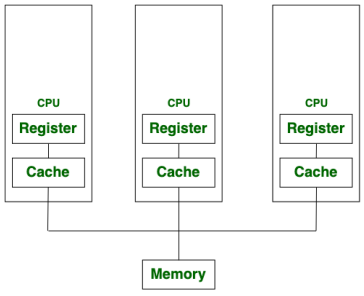


Multithreading

There could be multiple thread executing inside an application. Eg. We want to run word processor. This applciation/process has 3 threads. Graphic, responding to keystrokes and grammar check. Now first thread of word processor i.e. graphic happens for short span then switches to other thread in same process i.e. responding to keystrokes. Then it switches to grammar check and so on. This happens so fast that we cannot observe the delay. Therefore, a CPU can **execute parts of multiple parts of a program at the same time.**

▼ Multiprocessing

The availability of more than one processor per system, that can execute several set of instructions in parallel is known as **multiprocessing.**



Multiprocessing

Multitasking vs Multithreading vs Multiprocessing

Modern operating systems support multitasking (mainly preemptive multitasking), multithreading and multiprocessing (including symmetric multiprocessing and h...

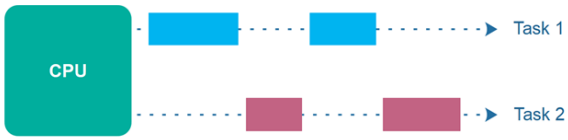
https://www.youtube.com/watch?v=Tn0u-IIBmtc&t=239s

▼ Concurrency and multithreading

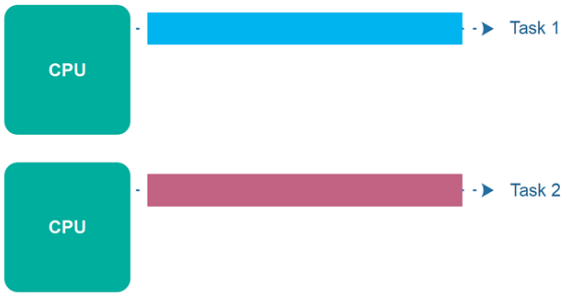
Concurrency is the **ability of a system** to handle multiple things by creating multiple execution units. Multithreading is the concept which helps **create multiple** execution units.

▼ Concurrency, Parallel execution and Parallelism

Concurrency is the concept of **executing two or more tasks at the same time** (in parallel). **Tasks may include methods (functions), parts of a program, or even other programs.**



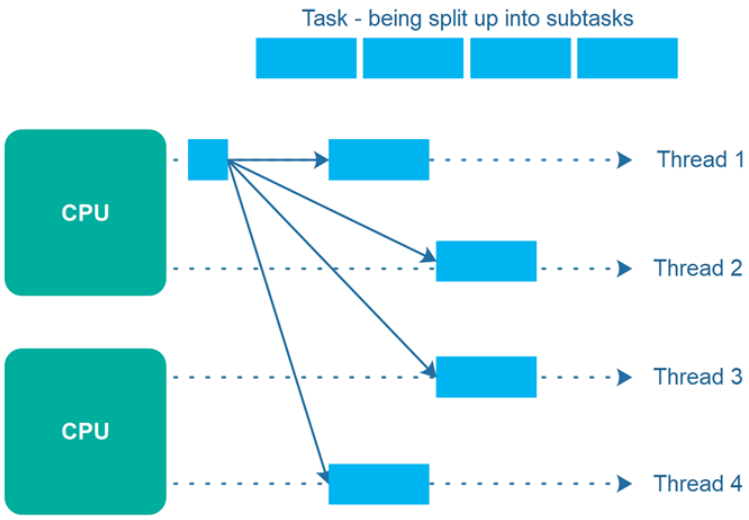
Parallel execution is when a computer has more than one CPU or CPU core, and makes progress on **more than one task** simultaneously. *However, parallel execution is not referring to the same phenomenon as parallelism. I will get back to parallelism later.*



Parallelism means that an application **splits its tasks up into smaller subtasks** which can be processed in parallel, for instance on multiple CPUs at the exact same time. Thus, parallelism does not refer to the same execution model as parallel concurrent execution - even if they may look similar on the surface.

To achieve true parallelism your application must have more than one thread running - and each thread must run on separate CPUs / CPU cores / graphics card GPU cores or similar.

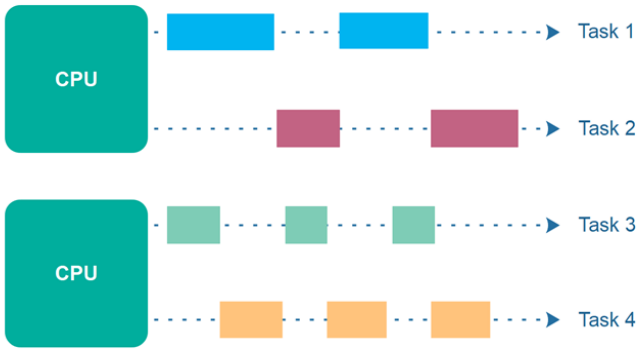
The diagram below illustrates a bigger task which is being split up into 4 subtasks. These 4 subtasks are being executed by 4 different threads, which run on 2 different CPUs. This means, that parts of these subtasks are executed concurrently (those executed on the same CPU), and parts are executed in parallel (those executed on different CPUs).



If instead the 4 subtasks were executed by 4 threads running on each their own CPU (4 CPUs in total), then the task execution would have been fully parallel. However, it is not always easy to break a task into exactly as many subtasks as the number of CPUs available. Often, it is easier to break a task into a number of subtasks which fit naturally with the task at hand, and then let the thread scheduler take care of distributing the threads among the available CPUs.

▼ Parallel and concurrent execution

It is possible to have parallel concurrent execution, where threads are distributed among multiple CPUs. Thus, the threads executed on the same CPU are executed **concurrently**, whereas threads executed on different CPUs are executed in **parallel**. The diagram below illustrates parallel concurrent execution.



▼ Advantages of multithreading

1. Better utilization of a single CPU.
2. Better utilization of multiple CPUs or CPU cores.
3. Better user experience with regards to responsiveness.
4. Better user experience with regards to fairness.

▼ Issue created due to multithreading

Multi-threading can **create other issues** like - If a thread reads a memory location while another thread writes to it....etc. Code executed by multiple threads accessing shared data need special attention. Errors arising from incorrect thread synchronisation can be very hard to detect, reproduce and fix.

▼ Context switching

When a CPU switches from executing one thread to executing another, the CPU needs to save the local data, program pointer etc. of the current thread, and load the local data, program pointer etc. of the next thread to execute. This switch is called a "context switch". The CPU switches from executing in the context of one thread to executing in the context of another.

Switching the cpu to another process requires performing a state of current process and a state restore of different process. This is called context switch.

Context switch is pure overhead.

https://www.youtube.com/watch?v=vTgccrbYHYs

▼ Concurrency Models

Concurrent systems can be implemented using different concurrency models. A concurrency model specifies **how threads in the the system collaborate** to complete the tasks they are given. Different concurrency models split the tasks in different ways, and the threads may communicate and collaborate in different ways.

▼ Shared State vs. Separate State

1. Shared state means that the different threads in the system will share some state among them. By state is meant **some data, typically one or more objects** or similar. When threads share state, problems like race conditions and deadlock etc. may occur. It depends on how the threads use and access the shared objects, of course.
2. Separate state means that the different threads in the system do not share any state among them. In case the different threads **need to communicate**, they do so either by **exchanging immutable objects** among them, or by sending copies of objects (or data) among them. Thus, when no two threads write to the same object (data / state), you can avoid most of the common concurrency problems.

▼ Parallel Workers

The first concurrency model is what I call the parallel workers model. Incoming jobs are assigned to different workers. Eg. in a car factory, each car would be produced by one worker.

The **advantage** of the parallel workers concurrency model is that it is easy to understand. To increase the parallelisation level of the application you just add more workers.

Disadvantage:

a. **Shared State Can Get Complex**: The threads need to access the shared data (when it exist) in a way that makes sure that changes by one thread are visible to the others (pushed to main memory and not just stuck in the CPU cache of the CPU executing the thread). Threads need to avoid race conditions, deadlock and many other shared state concurrency problems.

Additionally, part of the parallelization is lost when threads are waiting for each other when accessing the shared data structures.

b. **Stateless Workers**: Shared state can be modified by other threads in the system. Therefore workers must re-read the state every time they need it, to make sure they are working on the latest copy. This is true no matter whether the shared state is kept in memory or in an external database. **A worker** that does not keep state internally (but **re-reads it every time it is needed**) is called stateless .

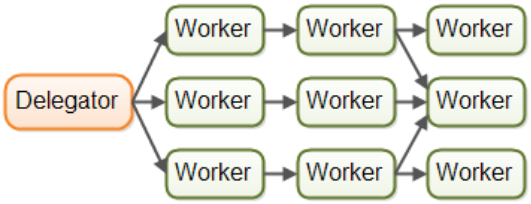
   Re-reading data every time you need it can get slow. Especially if the state is stored in an external database.

c. **Job execution order is nondeterministic:** There is no way to guarantee which jobs are executed first or last. Job A may be given to a worker before job B, yet job B may be executed before job A.

▼ Assembly line

The workers are organized like workers at an assembly line in a factory. Each worker only performs a part of the full job. When that part is finished the worker forwards the job to the next worker.

In reality, the jobs may not flow along a single assembly line. Since most systems can perform more than one job, jobs flows from worker to worker depending on what part of the job that needs to be executed next. In reality there could be multiple different virtual
assembly lines running on at the same time. This is how the job flow through an assembly line system might look in reality: Jobs may even be forwarded to more than one worker for concurrent processing.



▼ Functional Parallelism

Functional parallelism is a third concurrency model which is being talked about a lot these days (2015). The basic idea of functional parallelism is that you **implement** your program **using function calls.** When each function call can be executed independently, each function call can be executed on separate CPUs. That means, that an algorithm implemented functionally can be executed in parallel, on multiple CPUs.

▼ Creating and starting thread

https://www.youtube.com/watch?v=eQk5AWcTS8w

▼ Ways of starting a thread

Four ways to ways to specify what code a Java Thread should execute:

1. Java Class Implements Runnable
2. Anonymous Implementation of Runnable

3. Java Lambda Implementation of Runnable
4. Starting a Thread With a Runnable

▼ Thread start and run

Thread.start() vs Thread.run(): When a program calls the **start() method, a new thread is created** and then the run() method is executed. But if we directly call the run() method then no new thread will be created and **run() method will be executed as a normal method** call on the current calling thread itself and no multi-threading will take place.

Multiple invocation: In Java's multi-threading concept, another most important difference between start() and run() method is that we **can't call the start() method twice** otherwise it will throw an IllegalStateException whereas run() method can be called multiple times as it is just a normal method calling.

Threads may not execute sequentially, meaning the first thread called in main method may not be the first thread to write its name to System.out. This is because the threads are in principle executing in parallel and not sequentially. The JVM and/or operating system determines the order in which the threads are executed. This order does not have to be the same order in which they were started.

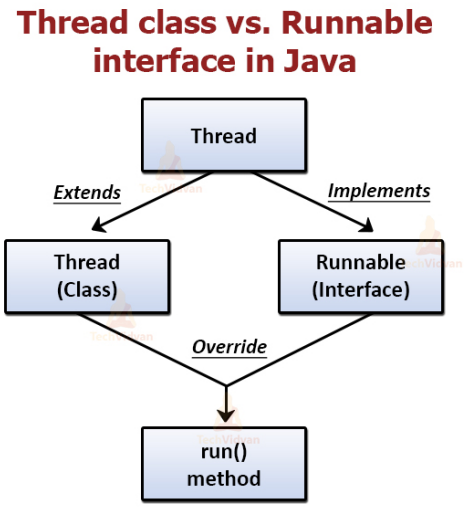
Runnable is Interface.

Thread is a class. *(class **Thread** implements Runnable)*

interface Runnable has a abstract method run().

▼ Java Runnable

Java runnable is **an interface used to execute code on a concurrent thread**. It is an interface which is implemented by any class if we want that the instances of that class should be executed by a thread. The runnable interface has an undefined method run() with void as return type, and it takes in no arguments.



▼ Callable vs Runnable

The Java `Callable` interface is similar to the Java `Runnable` interface, in that both of them represents a task that is intended to be executed concurrently by a separate thread.

A Java `Callable` is different from a `Runnable` in that the `Runnable` interface's `run()` method does not return a value, and it cannot throw checked exceptions (only `RuntimeException`s).

Additionally, a `Runnable` was originally designed for long running concurrent execution, e.g. running a network server concurrently, or watching a directory for new files. The `Callable` interface is more designed for one-off tasks that return a single result.

▼ Java Future

A Java *Future*, `java.util.concurrent.Future`, represents the result of an asynchronous computation. When the asynchronous task is created, a Java `Future` object is returned. This `Future` object functions as a handle to the result of the asynchronous task. Once the asynchronous task completes, the result can be accessed via the `Future` object returned when the task was started.

▼ Getting value from future

As mentioned earlier, a Java `Future` represents the result of an asynchronous task. To obtain the result, you call one of the two `get()` methods on the `Future`. The `get()` methods both return an `Object`, but the return type can also be a generic return type (meaning an object of a specific class, and not just an `Object`). Here is an example of obtaining the result from a Java `Future` via its `get()` method:

```
Future future = ... // get Future by starting async task

// do something else, until ready to check result via Future

// get result from Future
try {
    Object result = future.get();
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}
```

If you call the `get()` method before the asynchronous task has completed, the `get()` method will block until the result is ready.

There is a version of the `get()` method which can time out after an amount of time has passed which you can specify via method parameters. Here is an example of calling that `get()` version:

```
try {
    Object result =
        future.get(1000, TimeUnit.MILLISECONDS);
} catch (InterruptedException e) {
} catch (ExecutionException e) {
} catch (TimeoutException e) {
    // thrown if timeout time interval passes
    // before a result is available.
}
```

The example above waits for a maximum of 1000 milliseconds for the result to be available in the `Future`. If no result is available within 1000 milliseconds, a `TimeoutException` is thrown.

▼ Daemon threads

1. Daemon thread in Java is a low-priority thread that runs in the background to perform tasks such as garbage collection. Daemon thread in Java is also a service provider thread that provides services to the

user thread.

2. Its life depends on the mercy of user threads i.e. when all the user threads die, JVM terminates this thread automatically. In simple words, we can say that it provides services to user threads for background supporting tasks. It has no role in life other than to serve user threads.
3. They can not prevent the JVM from exiting when all the user threads finish their execution.
4. By default, the **main** thread is always **non-daemon** but for all the remaining threads, daemon nature will be inherited from parent to child.
5. Whenever the last non-daemon thread terminates, all the daemon threads will be terminated automatically.

Syntax: `t1.setDaemon(true);`

▼ Thread.sleep(millisec)

Need to be enclosed in try-catch block.

Thread. sleep() method throws **InterruptedException if a thread in sleep is interrupted by other threads**.

▼ Stop thread

```
Thread thread1 = new Thread();
thread1.stop();        //This exist but it is depricated
// and never used as we won't know where actually the thread
// has stopped. So, we create the myRunnable class which
// implements Runnable and include a boolean variable
// stopRequest which control the stopping of the thread.
```

MyRunnableWithStop.java

```
package com.java.concurrency;

public class MyRunnableWithStop implements Runnable
{
    ////////////////////////////////// Done to stop execution of thread.

    private boolean stopRequested = false;

    public synchronized void requestStop()
    {
        this.stopRequested = true;
    }

    private synchronized boolean isStopRequested()
    {
        return this.stopRequested;
    }

    //////////////////////////////////

    @Override
    public void run()
    {
        System.out.println("MyRunnable is Running parallelly.");

        while (!isStopRequested())
        {
            // keep doing what this thread should do.

            System.out.println("...");
```

```
            sleep(1000);
        }

        System.out.println("MyRunnable is stopped.");
    }

    public static void sleep(long millis)
    {
        try
        {
            Thread.sleep(millis);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

StoppingJavaThread.java

```
package com.java.concurrency;

public class StoppingJavaThread
{
    public static void main(String[] args)
    {
        MyRunnableWithStop stoppableRunnable = new MyRunnableWithStop();

        Thread t1 = new Thread(stoppableRunnable, "myThread ");

        try
        {
            t1.start();

            System.out.println(Thread.currentThread().getName() + " is pausing for 5 sec.");

            Thread.sleep(5000);

            System.out.println(Thread.currentThread().getName() + "'s pause finished.");


            System.out.println("\nRequesting stop");

            stoppableRunnable.requestStop();

            System.out.println("Stop requested\n");

        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

▼ Join thread

Join is a synchronization method that **blocks the calling thread** (that is, the thread that calls the method) until the thread whose Join method is called has completed. Use this method to ensure that a thread has been terminated. The caller will block indefinitely if the thread does not terminate.

Join()	Blocks the calling thread until the thread represented by this instance terminates, while continuing to perform standard COM and SendMessage pumping.

Join(TimeSpan)	Blocks the calling thread until the thread represented by this instance terminates or the specified time elapses, while continuing to perform standard COM and SendMessage pumping.
Join(Int32)	Blocks the calling thread until the thread represented by this instance terminates or the specified time elapses, while continuing to perform standard COM and SendMessage pumping.

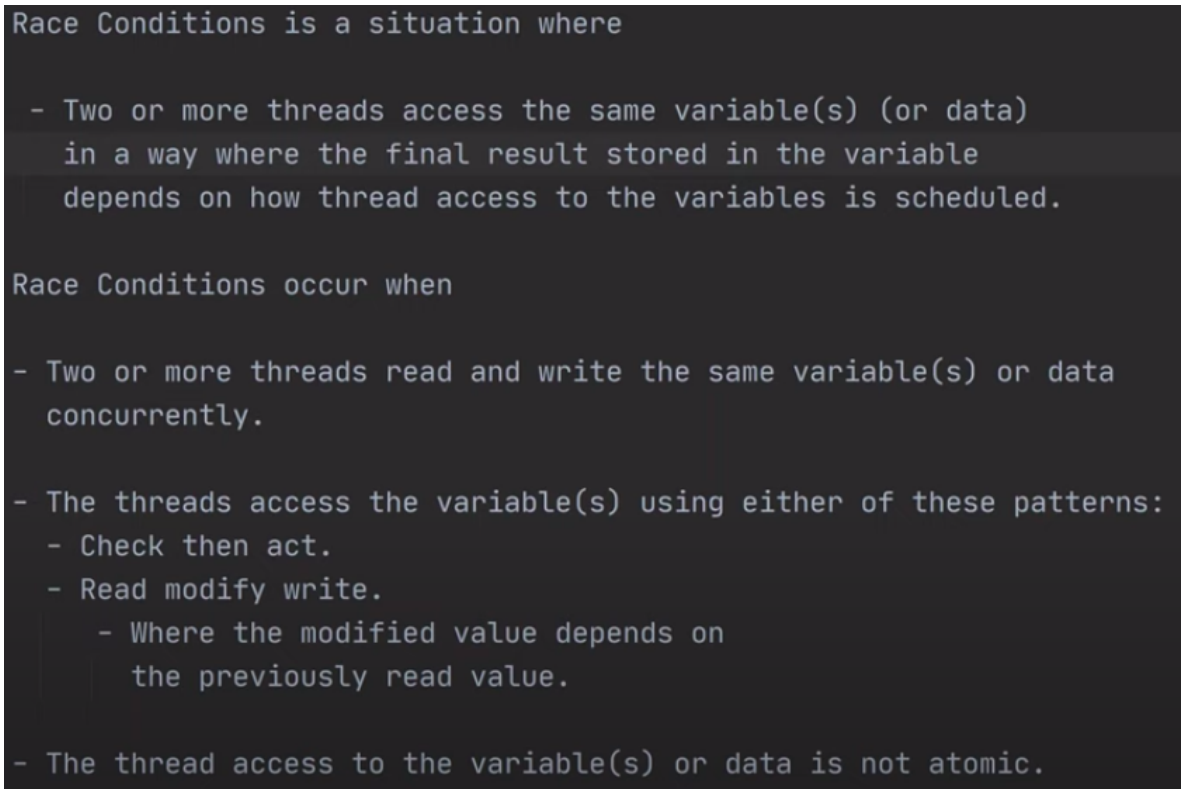
threadName.isAlive() returns boolean based on whether the thread is running or not currently.

See program: `WaitForDaemonThread.java`

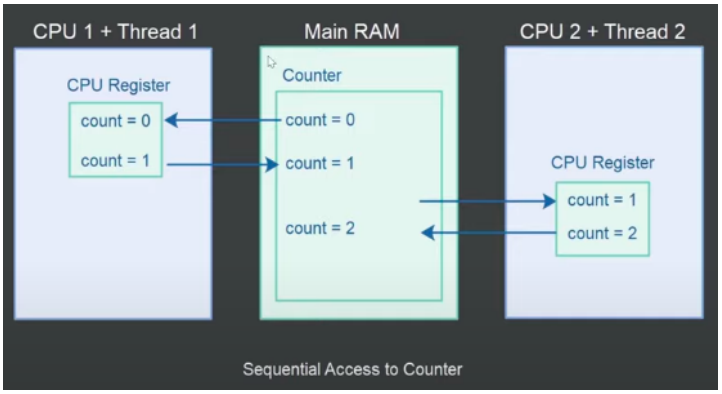
▼ Race condition

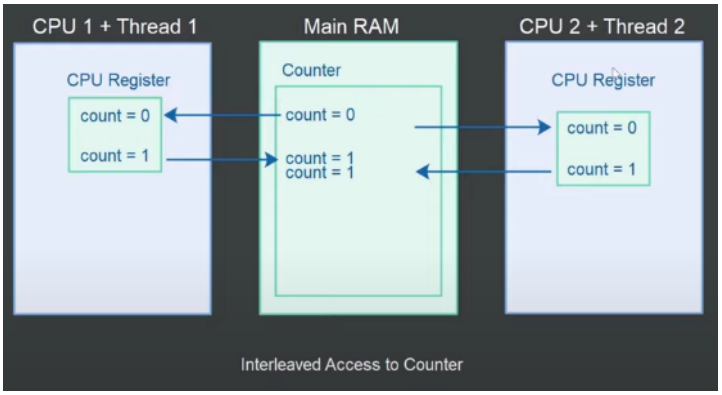
▼ Introduction

https://www.youtube.com/watch?v=RMR75VzYoos&list=PLL8woMHwr36EDxjUoCzboZjedsnhLP1j4&index=17



▼ Read modify write

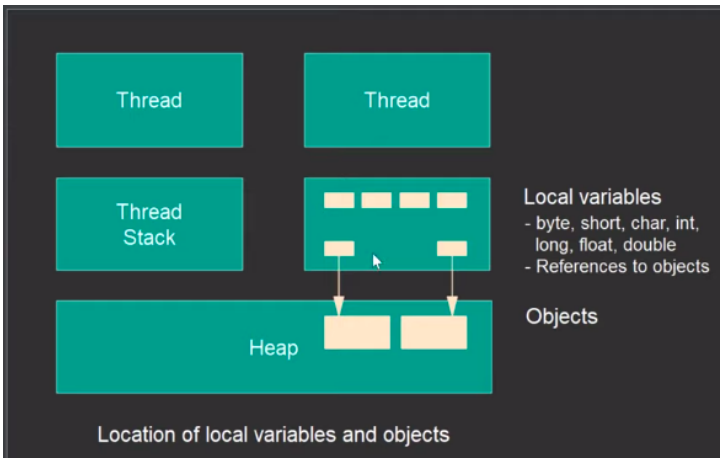




▼ Thread safety and shared resources

▼ Introduction

Code that is safe to call by multiple threads simultaneously is called *thread safe*. If a piece of code is thread safe, then **it contains no race conditions**. Race condition only occur when multiple threads update shared resources. Therefore it is important to know what resources Java threads share when executing.




```
class myRunnable implements Runnable
{
```

```
    private int count = 0;  //would be present in heap and shared

    @Override
    public void run()
    {
        MyObject myObject = new MyObject();  //present in thread's stack

        for (int i = 0; i < 1_00_000; i++)   //i also present in thread's stack
        {
            this.count++;
        }
        System.out.println(Thread.currentThread().getName()+":"+this.count);
    }
}
```

▼ Local Variables

Local variables are **stored in each thread's own stack**. That means that local variables are never shared between threads. That also means that all **local primitive variables are thread safe**. Here is an example of a thread safe local primitive variable:

```
public void someMethod(){

  long threadSafeInt = 0;

  threadSafeInt++;
}
```

▼ Local Object References

Local references to objects are a bit different. **The reference itself is not shared**. The object referenced however, is not stored in each threads's local stack. **All objects are stored in the shared heap**.

If an object created locally never escapes the method it was created in, it is thread safe. In fact you can also pass it on to other methods and objects as long as none of these methods or objects make the passed object available to other threads.

Here is an example of a thread safe local object:

```
public void someMethod(){

  LocalObject localObject = new LocalObject();

  localObject.callMethod();
  method2(localObject);
}

public void method2(LocalObject localObject){
  localObject.setValue("value");
}
```

The `LocalObject` instance in this example is not returned from the method, nor is it passed to any other objects that are accessible from outside the `someMethod()` method. Each thread executing the `someMethod()` method will create its own `LocalObject` instance and assign it to the `localObject` reference. Therefore the use of the `LocalObject` here is thread safe.

In fact, the whole method `someMethod()` is thread safe. Even if the `LocalObject` instance is passed as parameter to other methods in the same class, or in other classes, the use of it is thread safe.

The only exception is of course, if one of the methods called with the `LocalObject` as parameter, stores the `LocalObject` instance in a way that allows access to it from other threads.

▼ Object Member Variables

Object member variables (fields) are stored on the heap along with the object. Therefore, if two threads call a method on the same object instance and this method updates object member variables, the method is not thread safe. Here is an example of a method that is not thread safe:

```
public class NotThreadSafe{
    StringBuilder builder = new StringBuilder();

    public add(String text){
        this.builder.append(text);
    }
}
```

If two threads call the `add()` method simultaneously **on the same NotThreadSafe instance** then it leads to race conditions. For instance:

```
NotThreadSafe sharedInstance = new NotThreadSafe();

new Thread(new MyRunnable(sharedInstance)).start();
new Thread(new MyRunnable(sharedInstance)).start();

public class MyRunnable implements Runnable{
  NotThreadSafe instance = null;

  public MyRunnable(NotThreadSafe instance){
    this.instance = instance;
  }

  public void run(){
    this.instance.add("some text");
  }
}
```

Notice how the two `MyRunnable` instances share the same `NotThreadSafe` instance. Therefore, when they call the `add()` method on the `NotThreadSafe` instance it leads to race condition.

However, if two threads call the `add()` method simultaneously **on different instances** then it does not lead to race condition. Here is the example from before, but slightly modified:

```
new Thread(new MyRunnable(new NotThreadSafe())).start();
new Thread(new MyRunnable(new NotThreadSafe())).start();
```

Now the two threads have each their own instance of `NotThreadSafe` so their calls to the add method doesn't interfere with each other. The code does not have race condition anymore. So, even if an object is not thread safe it can still be used in a way that doesn't lead to race condition.

▼ The Thread Control Escape Rule

When trying to determine if your code's access of a certain resource is thread safe you can use the thread control escape rule:

```
If a resource is created, used and disposed within
the control of the same thread,
and never escapes the control of this thread,
the use of that resource is thread safe.
```

Resources can be any shared resource like an object, array, file, database connection, socket etc. In Java you do not always explicitly dispose objects, so "disposed" means losing or null'ing the reference to the object.

Even if the use of an object is thread safe, if that object points to a shared resource like a file or database, your application as a whole may not be thread safe. For instance, if thread 1 and thread 2 each create their own database connections, connection 1 and connection 2, the use of each connection itself is thread safe. But the use of the database the connections point to may not be thread safe. For example, if both threads execute code like this:

```
check if record X exists
if not, insert record X
```

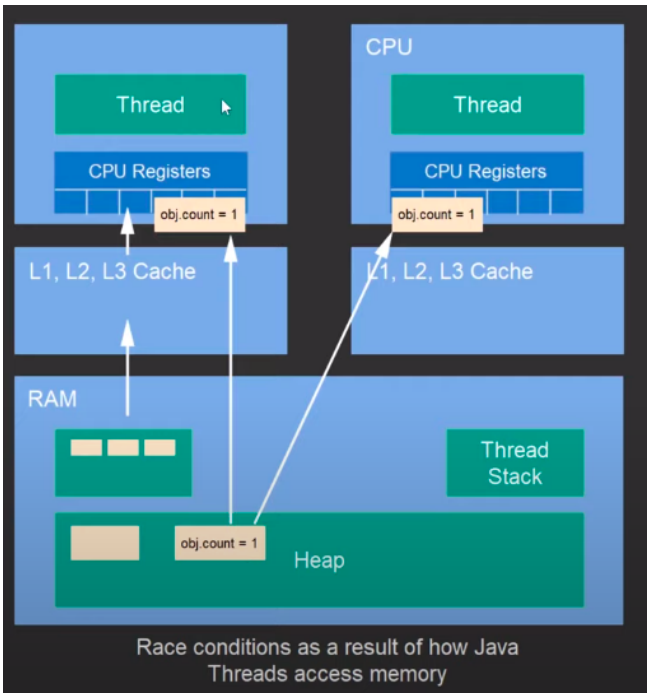
If two threads execute this simultaneously, and the record X they are checking for happens to be the same record, there is a risk that both of the threads end up inserting it. This is how:

```
Thread 1 checks if record X exists. Result = no
Thread 2 checks if record X exists. Result = no
Thread 1 inserts record X
Thread 2 inserts record X
```

This could also happen with threads operating on files or other shared resources. Therefore it is important to distinguish between whether an object controlled by a thread **is** the resource, or if it merely **references** the resource (like a database connection does).

▼ Java memory model

Java hardware architecture



▼ How to achieve thread safety?

Link: https://www.baeldung.com/java-thread-safety#:~:text=Atomic Objects,-safe%2C without using synchronization.

Thread safety can be acheived using the following methods:

- **Immutable implementation:** If we need to share state between different threads, we can create thread-safe classes by making them immutable i.e. by mentioning it **final**.
- We can easily create thread-safe collections by using the set of **synchronization wrappers** included within the collections framework.

```
Collection<Integer> syncCollection = Collections.synchronizedCollection(new ArrayList<>());
Thread thread1 = new Thread(() -> syncCollection.addAll(Arrays.asList(1, 2, 3, 4, 5, 6)));
Thread thread2 = new Thread(() -> syncCollection.addAll(Arrays.asList(7, 8, 9, 10, 11, 12)));
thread1.start();
thread2.start();
```

  This means that the methods can be accessed by only one thread at a time, while other threads will be blocked until the method is unlocked by the first thread.

- Java provides the ***java.util.concurrent*** package, which contains several concurrent collections, such as *ConcurrentHashMap*:

```
Map<String,String> concurrentMap = new ConcurrentHashMap<>();
concurrentMap.put("1", "one");
concurrentMap.put("2", "two");
concurrentMap.put("3", "three");
```

  Unlike their synchronized counterparts**, concurrent collections achieve thread-safety by dividing their data into segments**. In a *ConcurrentHashMap*, for instance, several threads can acquire locks on different

map segments, so multiple threads can access the *Map* at the same time.

Concurrent collections are much more performant than synchronized collections, due to the inherent advantages of concurrent thread access.

It's worth mentioning that **synchronized and concurrent collections only make the collection itself thread-safe and not the contents**.

- **Atomic classes allow us to perform atomic operations, which are thread-safe, without using synchronization**. An atomic operation is executed in one single machine level operation.

  It's also possible to achieve thread-safety using the set of atomic classes that Java provides, including *AtomicInteger*, *AtomicLong*, *AtomicBoolean*, and *AtomicReference*.

  Let's create a thread-safe implementation of the *Counter* class by using an *AtomicInteger* object:

```
public class Counter {

private final AtomicInteger counter = new AtomicInteger();

public void incrementCounter() {
        counter.incrementAndGet();
    }

public int getCounter() {
return counter.get();
    }
}
```

  Types of atomic objects in java. (Click on the links to know more)

  1. AtomicBoolean
  2. AtomicInteger
  3. AtomicLong
  4. AtomicReference
  5. AtomicStampedReference
  6. AtomicIntegerArray
  7. AtomicLongArray
  8. AtomicReferenceArray

- Synchronized methods
- Synchronized statements

```
synchronized(this){ ... }
```

- Other Objects as a Lock

  Caveats

- Volatile Fields
- Reentrant Locks
- Read/Write lock

▼ Synchronize

▼ synchronize(monitorObject)

It means that this block of code is `synchronized` meaning no more than one thread will be able to access the code inside that block.

Also `this` means you can synchronize on the current instance (obtain lock on the current instance).

This is what I found in Kathy Sierra's java certification book.

Because synchronization does hurt concurrency, you don't want to synchronize any more code than is necessary to protect your data. So if the scope of a method is more than needed, ***you can reduce the scope of the synchronized part to something less than a full method—to just a block****.*

Look at the following code snippet:

```
public synchronized void doStuff() {
    System.out.println("synchronized");
}
```

which can be changed to this:

```
public void doStuff() {
   //do some stuff for which you do not require synchronization
   synchronized(this) {
     System.out.println("synchronized");
     // perform stuff for which you require synchronization
   }
}
```

In the second snippet, the synchronization lock is only applied for that block of code instead of the entire method.

synchronized(X.class) vs synchronized(this)

synchronized(X. class) is **used to make sure that there is exactly one Thread in the block**. synchronized(this) ensures that there is **exactly one thread per instance of that particular class**. If this makes the actual code in the block thread-safe depends on the implementation. If mutate only state of the instance synchronized(this) is enough.

> Important program: `SynchronizeThisVsClass.java`

▼ Critical section throughput

For smaller critical sections making the whole critical section a synchronized block may work. But, for larger critical sections it may be beneficial to break the critical section into smaller critical sections, to allow multiple threads to execute each a smaller critical section. This may decrease contention on the shared resource, and thus increase throughput of the total critical section.

Here is a very simplified Java code example to show what I mean:

```
public class TwoSums {

    private int sum1 = 0;
```

```
    private int sum2 = 0;

    public void add(int val1, int val2){
        synchronized(this){
            this.sum1 += val1;
            this.sum2 += val2;
        }
    }
}
```

Notice how the `add()` method adds values to two different sum member variables. To prevent race conditions the summing is executed inside a Java synchronized block. With this implementation only a single thread can ever execute the summing at the same time.

However, since the two sum variables are independent of each other, you could split their summing up into two separate synchronized blocks, like this:

```
public class TwoSums {

    private int sum1 = 0;
    private int sum2 = 0;

    private Integer sum1Lock = new Integer(1);
    private Integer sum2Lock = new Integer(2);

    public void add(int val1, int val2){
        synchronized(this.sum1Lock){
            this.sum1 += val1;
        }
        synchronized(this.sum2Lock){
            this.sum2 += val2;
        }
    }
}
```

Now two threads can execute the `add()` method at the same time. One thread inside the first synchronized block, and another thread inside the second synchronized block. The two synchronized blocks are synchronized on different objects, so two different threads can execute the two blocks independently. This way threads will have to wait less for each other to execute the `add()` method.

▼ Sharing objects

https://www.youtube.com/watch?v=eKWjfZ-TUdo&list=PLL8woMHwr36EDxjUoCzboZjedsnhLP1j4&index=5

At 15:40

Don't **pass strings as the monitor objects** as we don't know how the compiler will consider 2 strings with same content.

▼ Synchronized Blocks in Lambda Expressions

It is even possible to use synchronized blocks inside a **Java Lambda Expression** as well as inside anonymous classes.

Here is an example of a Java lambda expression with a synchronized block inside. Notice that the synchronized block is synchronized on the class object of the class containing the lambda expression. It could have been synchronized on another object too, if that would have made more sense (given a specific use case), but using the class object is fine for this example.

```
import java.util.function.Consumer;

public class SynchronizedExample {

  public static void main(String[] args) {

    Consumer<String> func = (String param) -> {

      synchronized(SynchronizedExample.class)
      {
          System.out.println(Thread.currentThread().getName() + " step 1:
                              "+ param);
          try
          {
            Thread.sleep( (long) (Math.random() * 1000));
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }

          System.out.println(Thread.currentThread().getName() +
                      " step 2: " + param);
      }

    };


    Thread thread1 = new Thread(() -> {
        func.accept("Parameter");
    }, "Thread 1");

    Thread thread2 = new Thread(() -> {
        func.accept("Parameter");
    }, "Thread 2");

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

▼ Reentrance

Java monitors are reentrant means **java thread can reuse the same monitor for different synchronized methods if method is called from the method**.

There is a code called Reentrance.java and ReentranceMain.java.

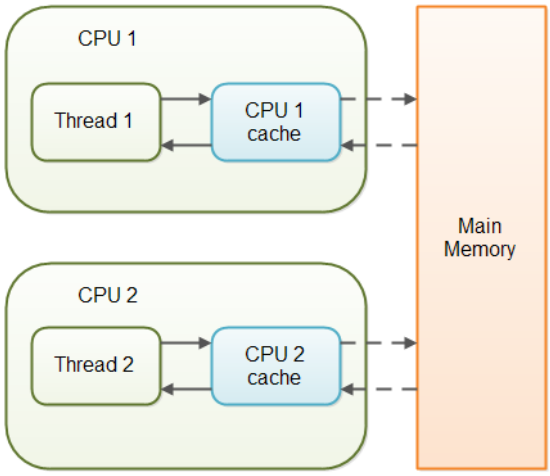
▼ Volatile

The Java `volatile` keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

Variable Visibility Problems

The Java `volatile` keyword guarantees visibility of changes to variables across threads. This may sound a bit abstract, so let me elaborate.

In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU cache while working on them, for performance reasons. If your computer contains more than one CPU, each thread may run on a different CPU. That means, that each thread may copy the variables into the CPU cache of different CPUs. This is illustrated here:



With non-volatile variables there are no guarantees about when the Java Virtual Machine (JVM) reads data from main memory into CPU caches, or writes data from CPU caches to main memory. This can cause several problems which I will explain in the following sections.

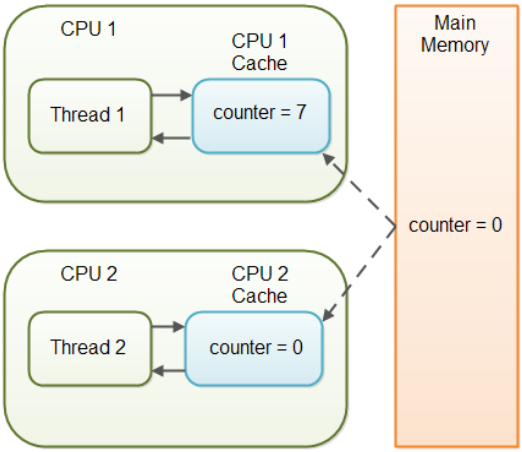
Imagine a situation in which two or more threads have access to a shared object which contains a counter variable declared like this:

```
public class SharedObject {

    public int counter = 0;
}
```

Imagine too, that only Thread 1 increments the `counter` variable, but both Thread 1 and Thread 2 may read the `counter` variable from time to time.

If the `counter` variable is not declared `volatile` there is no guarantee about when the value of the `counter` variable is written from the CPU cache back to main memory. This means, that the `counter` variable value in the CPU cache may not be the same as in main memory. This situation is illustrated here:



The problem with threads not seeing the latest value of a variable because it has not yet been written back to main memory by another thread, is called a "visibility" problem. The updates of one thread are not visible to other threads.

When is volatile Enough?

As I have mentioned earlier, if two threads are both reading and writing to a shared variable, then using the `volatile` keyword for that is not enough. You need to use a **synchronized** in that case to guarantee that the reading and writing of the variable is atomic. **Reading or writing a volatile variable does not block threads reading or writing**. For this to happen you must use the `synchronized` keyword around critical sections.

As an alternative to a `synchronized` block you could also use one of the many atomic data types found in the `java.util.concurrent` **package**. For instance, the `AtomicLong` or `AtomicReference` or one of the others.

In case only one thread reads and writes the value of a volatile variable and other threads only read the variable, then the reading threads are guaranteed to see the latest value written to the volatile variable. Without making the variable volatile, this would not be guaranteed.

The `volatile` keyword is guaranteed to work on 32 bit and 64 variables.

Performance Considerations of volatile

Reading and writing of volatile variables causes the variable to be read or written to main memory. **Reading from and writing to main memory is more expensive than accessing the CPU cache**. Accessing volatile variables also prevent instruction reordering which is a normal performance enhancement technique. Thus, you should only use volatile variables when you really need to enforce visibility of variables.

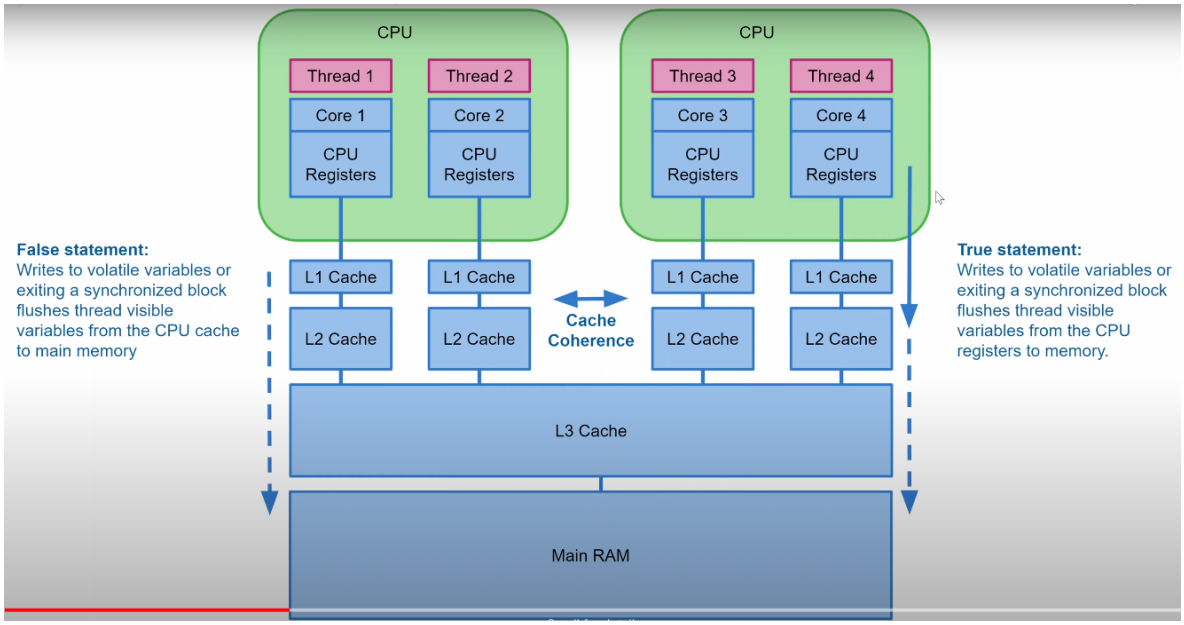
▼ Cache coherence

> When a Java thread writes to a volatile variable, or exits a synchronized block - that this flushes all variables visible to the thread from the CPU cache to main memory. This is not actually what happens.
>
> What actually happens is, that all variables visible to the thread which are stored in CPU registers will be flushed to main RAM (main memory). On the way to main RAM the variables may be stored in the CPU cache. The CPU / motherboard then uses its cache coherence methods to make sure that all other CPUs caches can see the variables in the first CPUs cache.

The hardware may even choose not to flush the variables all the way to main memory but only keep it in the CPU cache - until the CPU cache storing the variables is needed for other data. At that time the CPU cache can then be flushed to main memory. However, for the code running on the CPU this is not visible. As long as it gets the data it requests from any given memory address, it doesn't matter if the returned data only exists in the CPU cache, or whether it is also exists in main RAM.

You don't have to worry about how CPU cache coherence works. There is of course a little performance hit for this CPU cache coherence, but it's better than writing the variables all the way down to main RAM and back up the other CPU caches.

Below is a diagram illustrating what I said above. The red, dashed arrow on the left represents my false statement from other tutorials - that variables were flushed from CPU cache to main RAM. The arrow on the right represents what actually happens - that variables are flushed from CPU registers to the CPU cache.



Other cpus are able to see the data stored in other caches. Eg. core 1 accesses some value and changes it and if it stores it in the cache. Now, core 3 also want to access the same value then it can access it through the cache of core 1 even when the updated value is not flushed down to the main memory.

But we don’t know when the value would be flushed down from the cache to the main memory.

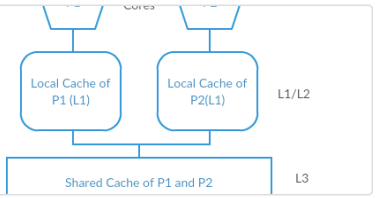
The volatile field is **needed to make sure that multiple threads always see the newest value**, even when the cache system or compiler optimizations are at work. Reading from a volatile variable always returns the latest written value from this variable.

▼ Cache and cooherence

Cache Coherency and Volatile Data

Before going into details about how to change the code to withstand the effects coming out of cache coherence, lets look at briefly what this cache coherence means. Cache coherency is the consistent nature of the shared data among multi cores. Cache coherency is a good

http://frozenlake-dul.blogspot.com/2015/11/cache-coherency-and-volatile-data.html



Lets consider the following example,

int X=10;

while(X==10){

/* Do something */

}

In the above code, the compiler may decide that X would not change during the run time. So the compiler might do an optimization to the code to omit X=10 considering it will not change. Because of this optimization, the system does not need to fetch X each iteration in the while loop

So the updated code would look something like this.

```
int X=10;
while( true ){
/* Do something */
}
```

So in the above code X=10 can be changed due to some other interruption as well. **But if we consider the cache coherency situation, X's value might get changed just because the value of X in the other core's local cache has changed already.** This is not done by the program but the hardware protocol.

So in order to avoid such issue, we need to instruct the compiler using a special keyword, not to optimize that variable in the code.

So that keyword is ***"volatile".*** So in above example we can use,

volatile int X=10;

Here are some important notes on using volatile.

- **If we apply volatile to a composite data type, the all the members of that composite element would become volatile.**
  - If we apply to a struct, all the data members in the struct would become volatile.
  - Also if we apply volatile to a struct member, the whole struct would become volatile.
- **Instances where volatile keyword would have no effect.**
  - If the data type is too long such that it cannot be fetched using a single instruction.
    - eg:- If applied to a struct, if the struct is longer than the maximum data length that can be fetched using a single instruction, depending on the architecture of the host machine, the struct would not be volatile.
- **The variables named as volatile would be fetched from the memory each time they are needed unless stored in a register. It would not be subject to certain optimizations by the compiler. So using volatile makes the program run a little slower.**

▼ False Sharing

False sharing in *Java* occurs when two threads running on two different CPUs write to two different variables which happen to be stored within the same CPU cache line. When the first thread modifies one of the variables - the whole CPU cache line is invalidated in the CPU caches of the other CPU where the other thread is running. This

means, that the other CPUs need to reload the content of the invalidated cache line - even if they **don't really need the variable that was modified within that cache line**.

https://www.youtube.com/watch?v=tLS85IfsbYE&list=PLL8woMHwr36EDxjUoCzboZjedsnhLP1j4&index=21

▼ Deadlock

Four condition that can cause deadlock to occur:

- Mutual Exclusion: one thread is used then it excludes all the other thread from using it
- No preemption: There could be no preemption. [Preemption: take action in order to prevent happening]
- Hold and wait
- Circular Wait

▼ Prevention

- Lock reordering
- Timeout backoff
- Deadlock detection

▼ Lock reordering

Deadlock occurs when multiple threads need the same locks but obtain them in different order.

If you make sure that all locks are always taken in the same order by any thread, deadlocks cannot occur. Look at this example:

```
Thread 1:
  lock A
  lock B

Thread 2:
   wait for A
   lock C (when A locked)

Thread 3:
   wait for A
   wait for B
   wait for C
```

If a thread, like Thread 3, needs several locks, it must take them in the decided order. It cannot take a lock later in the sequence until it has obtained the earlier locks.

For instance, neither Thread 2 or Thread 3 can lock C until they have locked A first. Since Thread 1 holds lock A, Thread 2 and 3 must first wait until lock A is unlocked. Then they must succeed in locking A, before they can attempt to lock B or C.

Lock ordering is a simple yet effective deadlock prevention mechanism. However, it can only be used if you know about all locks needed ahead of taking any of the locks. This is not always the case.

▼ Timeout backoff

Program: `DeadlockPreventionTimeout.java`

Deadlock prevention mechanism is to put a timeout on lock attempts meaning a thread trying to obtain a lock will only try for so long before giving up. If a thread does not succeed in taking all necessary locks within the given timeout, it will backup, free all locks taken, wait for a random amount of time and then retry. The **random amount of time** waited serves to give other threads trying to take the same locks a chance to take all locks, and thus let the application continue running without locking.

Here is an example of two threads trying to take the same two locks in different order, where the threads back up and retry:

```
Thread 1 locks A
Thread 2 locks B

Thread 1 attempts to lock B but is blocked
Thread 2 attempts to lock A but is blocked

Thread 1's lock attempt on B times out
Thread 1 backs up and releases A as well
Thread 1 waits randomly (e.g. 257 millis) before retrying.

Thread 2's lock attempt on A times out
Thread 2 backs up and releases B as well
Thread 2 waits randomly (e.g. 43 millis) before retrying.
```

If time is not randomised then the lock might quickly try to continuously aquire the lock and would lead to livelock.

What Is Livelock?

Livelock is another concurrency problem and is similar to deadlock. In livelock, **two or more threads keep on transferring states between one another** instead of waiting infinitely as we saw in the deadlock example. Consequently, the threads are not able to perform their respective tasks.

Progam at 8:30:

https://www.youtube.com/watch?v=6E3aYf3jXdk&list=PLL8woMHwr36EDxjUoCzboZjedsnhLP1j4&index=18

▼ Deadlock detection

Deadlock detection is a heavier deadlock prevention mechanism aimed at cases in which lock ordering isn't possible, and lock timeout isn't feasible.

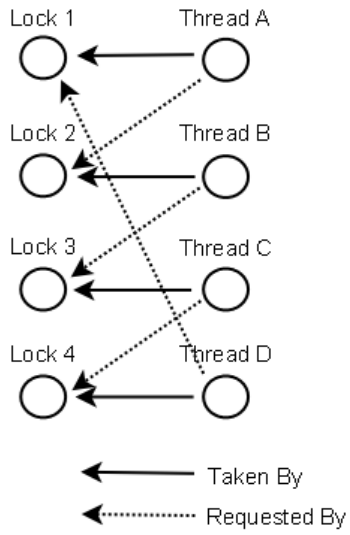
Every time a thread **takes** a lock it is noted in a data structure (map, graph etc.) of threads and locks. Additionally, whenever a thread **requests** a lock this is also noted in this data structure.

When a thread requests a lock but the request is denied, the thread can traverse the lock graph to check for deadlocks. For instance, if a Thread A requests lock 7, but lock 7 is held by Thread B, then Thread A can check if Thread B has requested any of the locks Thread A holds (if any). If Thread B has requested so, a deadlock has occurred (Thread A having taken lock 1, requesting lock 7, Thread B having taken lock 7, requesting lock 1).

Of course a deadlock scenario may be a lot more complicated than two threads holding each others locks. Thread A may wait for Thread B, Thread B waits for Thread C, Thread C waits for Thread D, and Thread D

waits for Thread A. In order for Thread A to detect a deadlock it must transitively examine all requested locks by Thread B. From Thread B's requested locks Thread A will get to Thread C, and then to Thread D, from which it finds one of the locks Thread A itself is holding. Then it knows a deadlock has occurred.

Below is a graph of locks taken and requested by 4 threads (A, B, C and D). A data structure like this that can be used to detect deadlocks.



So what do the threads do if a deadlock is detected?

One possible action is to release all locks, backup, wait a random amount of time and then retry. This is similar to the simpler lock timeout mechanism except threads only backup when a deadlock has actually occurred. Not just because their lock requests timed out. However, if a lot of threads are competing for the same locks they may repeatedly end up in a deadlock even if they back up and wait.

A better option is to determine or assign a priority of the threads so that only one (or a few) thread backs up. The rest of the threads continue taking the locks they need as if no deadlock had occurred. If the priority assigned to the threads is fixed, the same threads will always be given higher priority. To avoid this you may assign the priority randomly whenever a deadlock is detected.

▼ Lock

▼ Simple lock

```
public class Counter{

  private int count = 0;

  public int inc(){
    synchronized(this){
      return ++count;
    }
  }
}
```

Notice the `synchronized(this)` block in the `inc()` method. This block makes sure that only **one thread per instance** can execute the `return ++count` at a time. The code in the synchronized block could have been more

advanced, but the simple `++count` suffices to get the point across.

The `Counter` class could have been written like this instead, using a `Lock` instead of a synchronized block:

```
public class Counter{

  private Lock lock = new Lock();
  private int count = 0;

  public int inc(){
    lock.lock();
    int newCount = ++count;
    lock.unlock();
    return newCount;
  }
}
```

The `lock()` method locks the `Lock` instance so that all threads calling `lock()` are blocked until `unlock()` is executed.

▼ Synchronize vs Lock

```
1. Synchronized blocks must be contained within a single method.
   lock.lock() and lock.unlock() can be called from different methods.

2. lock.lock() and lock.unlock() provides the same visibility and
   happens before guarantees as entering and exiting a synchronized block

3. Synchronized blocks are always reentrant. Lock could decide not to be.

4. Synchronized blocks do not guarantee fairness. Locks can.
```

▼ tryLock() or tryLock(long time, TimeUnit unit)

Acquires the lock only if it is not held by another thread at the time of invocation.

Acquires the lock if it is not held by another thread and returns immediately with the value `true`, setting the lock hold count to one. Even when this lock has been set to use a fair ordering policy, a call to `tryLock()` *will* immediately acquire the lock if it is available, whether or not other threads are currently waiting for the lock. This "barging" behavior can be useful in certain circumstances, **even though it breaks fairness**. If you want to honor the fairness setting for this lock, then use `tryLock(0, TimeUnit.SECONDS)` which is almost equivalent (it also detects interruption).

▼ lock() vs tryLock()

boolean tryLock() – this is a non-blocking version of lock() method; it attempts to acquire the lock immediately, **return true if locking** succeeds. boolean tryLock(long timeout, TimeUnit timeUnit) – this is similar to tryLock(), except it waits up the given timeout before giving up trying to acquire the Lock.

▼ ReentrantLock()

A reentrant lock is **a mutual exclusion mechanism that allows threads to reenter into a lock on a resource (multiple times) without a deadlock situation**. A thread entering into the lock increases the hold count by one every time.

Calender example: At 7:10

You are keeping a lock in your addition method.

Now that addition is been used in the main calculator method which is also surrounded by locks.

```
public void add(double value) {
    try {
        lock.lock();
        this.result += value;
    } finally {
        lock.unlock();
    }
}

public void subtract(double value) {
    try {
        lock.lock();
        this.result -= value;
    } finally {
        lock.unlock();
    }
}
```

```
public void calculate(Calculation ... calculations) {
    try {
        lock.lock();

        for(Calculation calculation : calculations) {
            switch(calculation.type) {
                case Calculation.ADDITION   : add     (calculation.value); break;
                case Calculation.SUBTRACTION: subtract(calculation.value); break;
            }
        }
    } finally {
        lock.unlock();
    }
}
```

https://www.youtube.com/watch?v=MWlqrLiscjQ&list=PLL8woMHwr36EDxjUoCzboZjedsnhLP1j4&index=12

```
ReentrantLock lock = new ReentrantLock();
int     holdCount          = lock.getHoldCount();
int     queueLength        = lock.getQueueLength();
boolean hasQueueThisThread = lock.hasQueuedThread(Thread.currentThread());
boolean hasQueuedThreads   = lock.hasQueuedThreads();
boolean isFair             = lock.isFair();
boolean isLocked           = lock.isLocked();
boolean isHeldByCurrentThread = lock.isHeldByCurrentThread();
```

For fairness:

```
ReentrantReadWriteLock readWriteLock1 = new ReentrantReadWriteLock(true);
```

▼ Reentrant lockout

Reentrance lockout is a situation similar to **deadlock** and **nested monitor lockout**. Reentrance lockout is also covered in part in the texts on **Locks** and **Read / Write Locks**.

Reentrance lockout may occur if a thread reenters a Lock, ReadWriteLock or some other synchronizer that is not reentrant. Reentrant means that a thread that already holds a lock can retake it. Java's synchronized blocks are reentrant. Therefore the following code will work without problems:

```
public class Reentrant{

  public synchronized outer(){
    inner();
  }

  public synchronized inner(){
    //do something
  }
}
```

Notice how both `outer()` and `inner()` are declared synchronized, which in Java is equivalent to a `synchronized(this)` block. If a thread calls `outer()` there is no problem calling inner() from inside `outer()`, since both methods (or blocks) are synchronized on the same monitor object ("this"). If a thread already holds the lock on a monitor object, it has access to all blocks synchronized on the same monitor object. This is called reentrance. The thread can reenter any block of code for which it already holds the lock.

The following `Lock` implementation is not reentrant:

```
public class Lock{

  private boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      wait();
    }
    isLocked = true;
  }

  public synchronized void unlock(){
```

```
    isLocked = false;
    notify();
  }
}
```

If a thread calls `lock()` twice without calling `unlock()` in between, the second call to `lock()` will block. A reentrance lockout has occurred.

To avoid reentrance lockouts you have two options:

1. Avoid writing code that reenters locks
2. Use reentrant locks

Which of these options suit your project best depends on your concrete situation. Reentrant locks often don't perform as well as non-reentrant locks, and they are harder to implement, but this may not necessary be a problem in your case. Whether or not your code is easier to implement with or without lock reentrance must be determined case by case.

▼ lockInterruptibly()

`lockInterruptibly()` may block if the the lock is already held by another thread and will wait until the lock is aquired. This is the same as with regular `lock()`. But if another thread interrupts the waiting thread `lockInterruptibly()` will throw `InterruptedException`.

▼ Read write lock

Read Access	If no threads are writing, and no threads have requested write access.
Write Access	If no threads are reading or writing.

If a thread wants to read the resource, it is okay as long as no threads are writing to it, and no threads have requested write access to the resource. By up-prioritizing write-access requests we assume that write requests are more important than read-requests. Besides, if reads are what happens most often, and we did not up-prioritize writes, **starvation** could occur. Threads requesting write access would be blocked until all readers had unlocked the `ReadWriteLock`. If new threads were constantly granted read access the thread waiting for write access would remain blocked indefinately, resulting in **starvation**. Therefore a thread can only be granted read access if no thread has currently locked the `ReadWriteLock` for writing, or requested it locked for writing.

▼ Fairness

Java's synchronized blocks makes no guarantees about the sequence in which threads trying to enter them are granted access. Therefore, if many threads are constantly competing for access to the same synchronized block, there is a risk that one or more of the threads are never granted access - that access is always granted to other threads. This is called starvation. To avoid this a `Lock` should be fair. Since the `Lock` implementations shown in this text uses synchronized blocks internally, they do not guarantee fairness.

```
Lock lock = new ReentrantLock(true);
//true is set to use a fair ordering policy in the lock
```

> tryLock() **does not provide any guarantee of fairness** even if ReentrantLock(true) is set.

▼ Happens before

Happens-before is not any keyword or object in the Java language, it is simply **a discipline or set of restrictions put into place** so that in a multi-threading environment the reordering of the surrounding instructions does not result in a code that produces incorrect output.

▼ Blocking queue

▼ BlockingQueue methods

	Throws exception	Special value	Blocks	Times out (returns boolean)
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	*not applicable*	*not applicable*

`drainTo(Collection<? super E> c)`	Removes all available elements from this queue and adds them to the given collection.
`drainTo(Collection<? super E> c, int maxElements)`	Removes at most the given number of available elements from this queue and adds them to the given collection. **Remove from beginning (from where we poll).**

- ***offer("3", 1000, TimeUnit.MILLISECONDS)***

  Throws:

  `InterruptedException` - if interrupted while waiting

  `ClassCastException` - if the class of the specified element prevents it from being added to this queue

  `NullPointerException` - if the specified element is null

  `IllegalArgumentException` - if some property of the specified element prevents it from being added to this queue

▼ ArrayBlockingQueue vs LinkedBlockingQueue

ArrayBlockingQueue	LinkedBlockingQueue
It stores the elements internally in an array.	It stores the elements internally in linked nodes.
ArrayBlockingQueue is bounded which means the size will never change after its creation.	**LinkedBlockingQueue** is optionally bounded which means it can optionally have an upper bound if desired. If no upper bound is specified, **Integer.MAX_VALUE** is used as the upper bound.
It has lower throughput than linked nodes queues.	It has a higher throughput than array-based queues.
It uses the single-lock double condition algorithm. It means that producer and consumer share a single lock.	It uses two lock queue algorithms and it has two lock conditions putLock and takeLock for inserting and removing elements respectively from the Queue.
ArrayBlockingQueue always holds an object array.	LinkedBlockingQueue is a linked node with an object with three object fields.

LinkedBlockingQueue uses **Double lock queue**, A separate lock is used for **put and offer** methods, and a take lock is used for **take and poll**. In addition, because there are two locks, the counter count is updated with

Atomic variables, which avoids operating two locks at the same time to update data. There is a problem of visibility, because the two locks are independent, that is, **put and take use different synchronization blocks.**

How can the put data be made visible in take?

According to the documentation on the Java official website, the visibility is only based on the lock of the same monitor. After one thread is released, the other thread can obtain the lock to gain visibility, but here it uses the enhanced semantics of volatile to ensure visibility, The put operation will update the count variable modified with volatile, and then if a reader thread enters, if the count variable modified by volatile is accessed first, then volatile write has a happened-before relationship for reading, which means that as long as the volatile variable is accessed, the previous data modified by threads of different locks will be forced to refresh to the main cache so that the reader thread can read it, but this only guarantees visibility. **How is atomicity guaranteed**? Here it takes advantage of the characteristics of the queue, The characteristic of the queue is that the head node goes out of the queue and the end node goes into the queue, which means that no two threads modify the same node at the same time at any time, thus cleverly avoiding this problem.

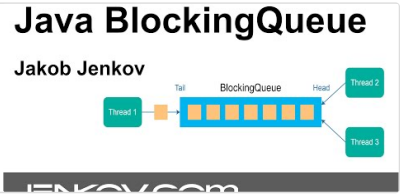
▼ Array blocking queue

Method	Description
put(E e)	Inserts the specified element at the tail of this queue, **waiting** for space to become available if the queue is full.
add(E e) [mostly throws exception]	Inserts the specified element at the tail of this queue if it is possible to do so immediately without exceeding the queue's capacity, returning true upon success and throwing an **IllegalStateException if this queue is full**.
offer(E e) [mostly results in true or false]	Inserts the specified element at the tail of this queue if it is possible to do so immediately without exceeding the queue's capacity, returning true upon success and false if this queue is full.
take()	Retrieves and removes the head of this queue, **waiting** if necessary until an element becomes available.
poll() [mostly passes null if empty]	Retrieves and removes the head of this queue, or **returns null if this queue is empty**.
remove()	Gives exception if any element not present. Method from the parent class AbstractQueue.
remove(Object o)	Removes a single instance of the specified element from this queue, if it is present. Returns **true** if object deleted else **false**.
peek()	Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
element()	Method from parent class AbstractQueue. Retrieves, but does not remove, the head of this queue. This method differs from `peek` only in that it throws an exception if this queue is empty.

Java BlockingQueue

The Java BlockingQueue interface represents a queue which can block threads inserting elements into the queue if the BlockingQueue is full, or thread removin...

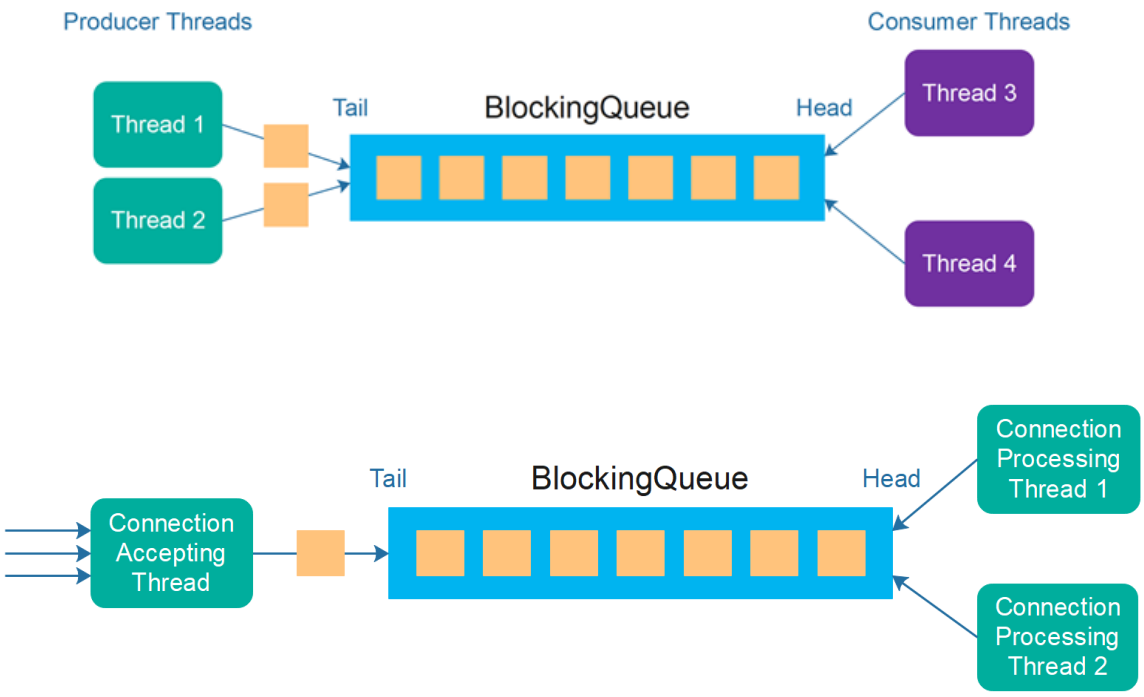
https://www.youtube.com/watch?v=d3xb1Nj88pw



▼ Linked blocking queue

Main difference: If you specify size you will get fix remainingCapacity of your blocking queue while if you don't you will get very high remainingCapacity.

▼ Producer Consumer Pattern



Reduce Foreground thread latency

In some systems you have a single foreground thread which has the communication with the outside world. **In a server it could be the thread accepting the incoming connections from clients**. In a desktop app that could be the UI thread.

In order to not make the foreground thread busy with tasks - whatever tasks the foreground thread receives from the outside world are **offloaded to one or more background threads**. In a server it could be processing the data that is received via the inbound connections.

Load balance between worker threads

Another type of use case for the producer consumer pattern is to load balance work between a set of worker threads. Actually, this load balancing happens pretty much automatically, as long as the worker threads take new task objects from the queue as soon as they have time to process them. This will result in load balancing the tasks between the worker threads.

Backpressure management

If the queue between the producer and consumer threads is a **Java BlockingQueue**, then you can use the queue for backpressure management. This is another built-in feature of the producer consumer pattern.

Backpressure means, that if the producer thread(s) produce more work than the consumer threads are able to handle - the tasks will queue up in the queue. At some point the BlockingQueue will become full, and the producer threads will be blocked trying to insert new tasks / work objects into the queue. This phenomenon is called *backpressure*. **The system presses back against the producers - towards the beginning of the "pipeline" - preventing more work from coming in.**

The backpressure will "spill out" of the queue, and slow down the producer thread(s). Thus, they too could propagate the pressure back up the work pipeline, if there are any earlier steps in the total pipeline.

▼ Semaphore

A counting semaphore. Conceptually, a semaphore maintains a set of permits. Each

`acquire()` blocks if necessary until a permit is available, and then takes it. Each `release()` adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the **Semaphore just keeps a count of the number available and acts accordingly.**

Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource.

See program: `SemaphoreMain.java`

Improper usage of semaphore causes deadlock and starvation

https://www.youtube.com/watch?v=uP9ICOH0Xsw

▼ ThreadLocal

This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable.

ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).

For example, the class below generates unique identifiers local to each thread. A thread's id is assigned the first time it invokes `ThreadId.get()` and remains unchanged on subsequent calls.

Operations:

```
ThreadLocal<String> threadLocal = new ThreadLocal<>();

Thread thread1 = new Thread(() ->
{
    try
    {
        threadLocal.set("thread1");

        Thread.sleep(3000);

        System.out.println(threadLocal.get());
        //get just shows value it does not removes it

        threadLocal.remove();    //it only removes value it does not return anything

        System.out.println(threadLocal.get());

    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
});
```

Have done a program called `ThreadLocalMain.java`.

https://www.youtube.com/watch?v=a_BoqsnVR2U&list=PLL8woMHwr36EDxjUoCzboZjedsnhLP1j4&index=10

▼ Thread Pools

▼ Introduction

A *thread pool* is a pool threads that can be "reused" to execute tasks, so that **each thread may execute more than one task**. A thread pool is an alternative to creating a new thread for each task you need to execute.

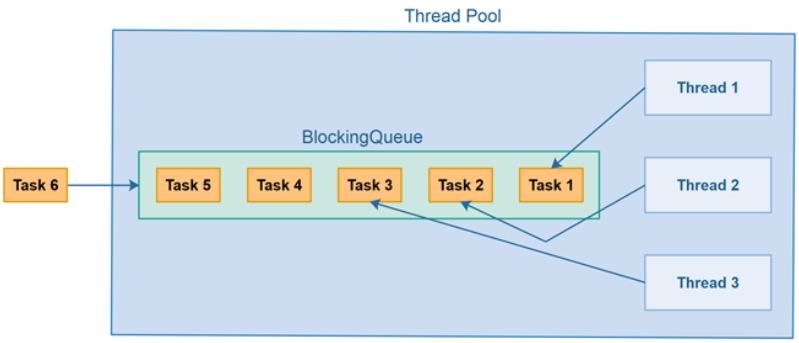
Creating a new thread comes with a performance **overhead compared to reusing a thread** that is already created. That is why reusing an existing thread to execute a task can result in a higher total throughput than creating a new thread per task.

Additionally, using a thread pool can make it easier to control how many threads are active at a time. Each thread consumes a certain amount of computer resources, such as memory (RAM), so if you have too many threads active at the same time, the total amount of resources (e.g. RAM) that is consumed may cause the computer to slow down - e.g. if so much RAM is consumed that the operating system (OS) starts swapping RAM out to disk.

Java already contains a built-in thread pool - the **Java ExecutorService** - so you can use a thread pool in Java without having to implement it yourself.

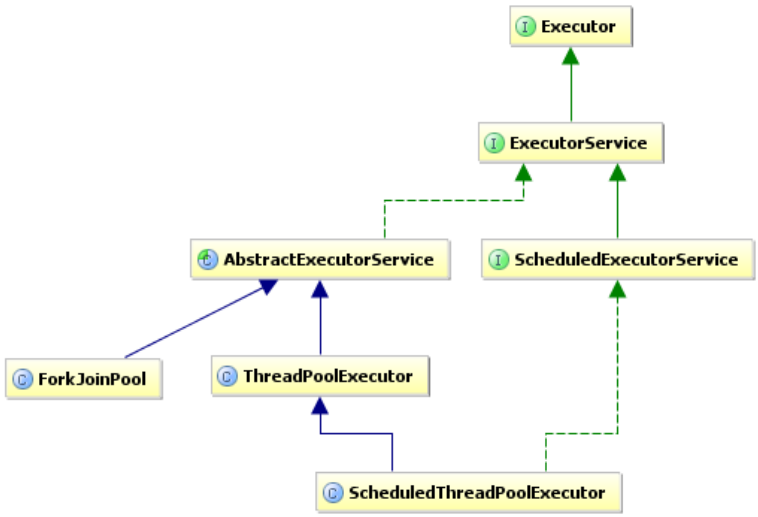
▼ How a thread pool works?

Instead of starting a new thread for every task to execute concurrently, the task can be passed to a thread pool. As soon as the pool has any idle threads the task is assigned to one of them and executed. Internally the tasks are inserted into a **Blocking Queue** which the threads in the pool are dequeuing from. When a new task is inserted into the queue one of the idle threads will dequeue it successfully and execute it. The rest of the idle threads in the pool will be blocked waiting to dequeue tasks.



▼ Executor framework hierarchy

Here, Executors is a separate **class** that includes Factory and utility methods for `Executor`, `ExecutorService`, `ScheduledExecutorService`, `ThreadFactory`, and `Callable` classes defined in this package.




```
ExecutorService executorService = Executors.newFixedThreadPool(3);
executorService.execute(newRunnable("Task 1"));
//new runnable created by me which implements Runnable
executorService1.shutdown();
```

```
ExecutorService threadPoolExecutor = new
                    ThreadPoolExecutor(corePoolSize,        //starts with pool size
                    maxPoolSize,                            //extend to max this much size
                    keepAliveTime,     //will be alive for this much idle time before terminating
                    TimeUnit.MILLISECONDS,
                    new ArrayBlockingQueue<Runnable>(128));   //queue in which tasks are stored

threadPoolExecutor.execute(new Runnable(...Implementation of run...))

threadPoolExecutor.shutdown();       //if not done then it would continue executing
```

▼ Methods in Executor class

newFixedThreadPool(int nThreads)	Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.
newSingleThreadExecutor()	Similar to newFixedThreadPool(1).
submit(**Runnable** task)	Submits a Runnable task for execution and returns a Future representing that task.

Think of a *Future* **as an object that holds the result** – it may not hold it right now, but it will do so in the future (once the Callable returns). Thus, a Future is basically one way the main thread can keep track of the progress and result from other threads.

`ExecutorService executorService = Executors.newFixedThreadPool(3)`

executorService.execute(*newRunnable*(**"Task 1"**));	Executes the given command at some time in the future. The command may execute in a new thread, in a pooled thread, or in the calling thread, at the discretion of the Executor implementation.

▼ ThreadPoolExecutor and ScheduledThreadPoolExecutor

ScheduledThreadPoolExecutor is a ThreadPoolExecutor that can additionally schedule commands to run after a given delay, or to execute periodically.

▼ ForkJoinPool

Similar to ExecutorService the differences present are:

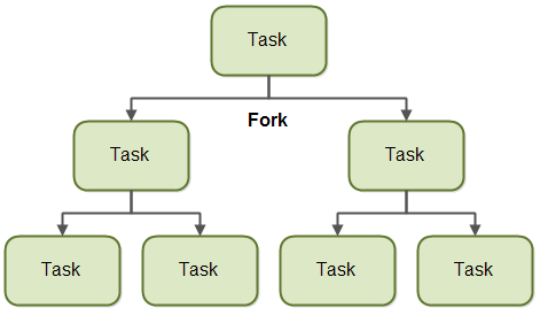
- Task predicting sub-tasks a.k.a. fork join
- Pre-thread queuing and work stealing

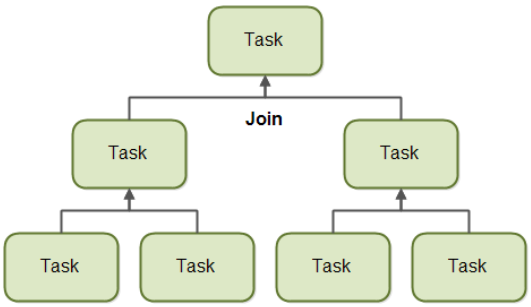
https://www.youtube.com/watch?v=5wgZYyvIVJk

Program: `ForkJoinFibonacci.java`

The prorgam breaks the work down into subtasks, and schedules these subtasks for execution using their `fork()` method.

Additionally, this example then receives the result returned by each subtask by calling the `join()` method of each subtask. The subtask results are merged into a bigger result which is then returned. This kind of joining / mergining of subtask results may occur recursively for several levels of recursion.





Fork join pool should avoid

- Avoid synchronization
- Do not use shared variables across tasks
- Do not perform Blocking IO operations
- Are pure functions
- Are isolated

▼ Code

```
package com.java.concurrency;

import java.util.concurrent.ForkJoinPool;

import java.util.concurrent.RecursiveTask;

public class ForkJoinFibonacci extends RecursiveTask<Integer>
{
    int n;

    ForkJoinFibonacci(int num)
    {
        this.n = num;
    }

    @Override
    protected Integer compute()
    {
        try
        {
            if (n <= 1)
            {
                return n;
            }

            ForkJoinFibonacci forkJoinFibonacci1 = new ForkJoinFibonacci(n - 1);

            forkJoinFibonacci1.fork();

            ForkJoinFibonacci forkJoinFibonacci2 = new ForkJoinFibonacci(n - 2);

            forkJoinFibonacci2.fork();

            return (forkJoinFibonacci1.join() + forkJoinFibonacci2.join());
        }
        catch (Exception ex)
```

```
        {
            ex.printStackTrace();
        }
        return null;
    }

    public static void main(String[] args)
    {
        try
        {
            ForkJoinPool pool = new ForkJoinPool();

            int ans = pool.invoke(new ForkJoinFibonacci(6));

            System.out.println("Inside ForkJoin for number 6:  " + ans);
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
//OUTPUT: Inside ForkJoin for number 6:  8
```

▼ Amdahl’s law

Suppose, Moni have to attend an invitation. Moni’s another two friend Diya and Hena are also invited. There are conditions that all three friends have to go there separately and all of them have to be present at door to get into the hall. Now Moni is coming by car, Diya by bus and Hena is coming by foot. Now, how fast Moni and Diya can reach there it doesn’t matter, they have to wait for Hena. So to speed up the overall process, we need to concentrate on the performance of Hena other than Moni or Diya.

This is actually happening in Amdahl’s Law. It relates the improvement of the system’s performance with the parts that didn’t perform well, like we need to take care of the performance of that parts of the systems. This law often used in parallel computing to predict the theoretical speedup when using multiple processors.

Over here Hena is considered as **serial processing** which can’t be optimised while the others can be called as the **parallel processsors** which can be optimised by adding more processors.

Eg. Imagine a program that processes files from disk. A small part of that program may scan the directory and create a list of files internally in memory. After that, each file is passed to a separate thread for processing. The part that scans the directory and creates the file list **cannot be parallelized**, but processing the files **can be parallelized**. Hence, on increasing core would affect the processing of files parallelly but won’t affect the serial process. Therefore, we get the following formula.

Formula

Amdahl’s Law can be expressed in mathematically as follows −

$$SpeedupMAX = \frac{1}{(1-p) + (p/s)}$$

SpeedupMAX = maximum performance gain

s = performance gain factor of p after implement the enhancements OR **number of cores that are present.**

p = the part which performance needs to be improved.

Let’s take an example, if the part that can be improved is 30% of the overall system and its performance can be doubled for a system, then −

SpeedupMAX = 1/((1-0.30)+(0.30/2))

= 1.18

Now, in another example, if the part that can be improved is 70% of the overall system and its performance can be doubled for a system, then −

SpeedupMAX= 1/((1-0.70)+(0.70/2))

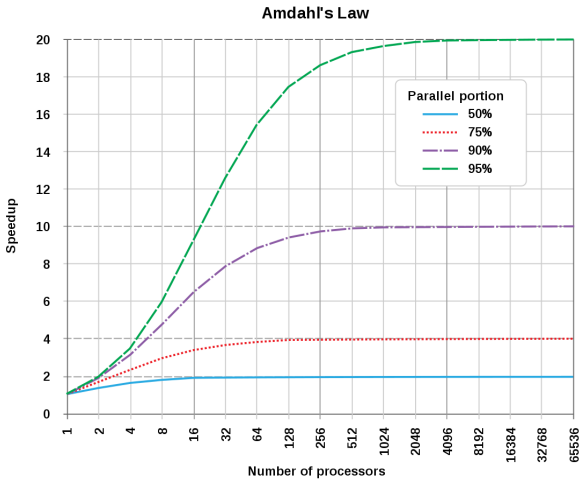
= 1.54

So, we can see, if 1-p can’t be improved, the overall performance of the system cannot be improved so much. So, if 1-p is 1/2, then speed cannot go beyond that, no matter how many processors are used.

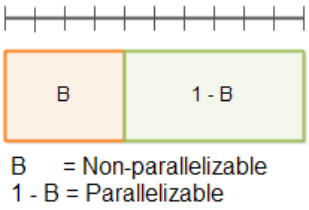
Multicore programming is most commonly used in signal processing and plant-control systems. In signal processing, one can have a concurrent system that processes multiple frames in parallel. the controller and the plant can execute as two separate tasks, in plant-control systems.

Multicore programming helps to split the system into multiple parallel tasks, which run simultaneously, speeding up the overall execution time.

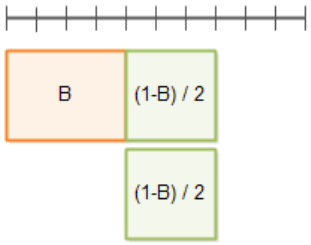
After increasing processors to a particular level won’t increase the speedup that extremely.

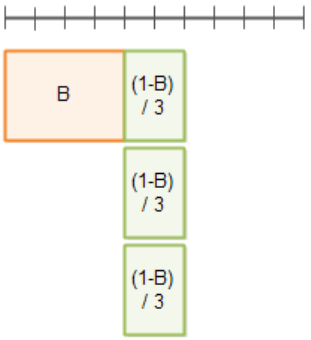


Illustration



B = Non-parallelizable
1 - B = Parallelizable





▼ AtomicInteger

```
public class AtomicInteger
extends Number
implements Serializable
```

The `AtomicInteger` class provides you with a `int` variable which can be read and written **atomically**, and which also contains advanced atomic operations like `compareAndSet()`.

The net result is that it's impossible for two cores to "successfully" perform compareAndSet operations on the same storage at the same time. Instead, hardware will delay one of the actions so that they occur sequentially.

An `int` value that may be updated atomically. See the `java.util.concurrent.atomic` package specification for description of the properties of atomic variables. An `AtomicInteger` is used in applications such as atomically incremented counters, and **cannot be used as a replacement** for an `Integer`. However, this class does extend `Number` to allow uniform access by tools and utilities that deal with numerically-based classes.

> Primitive types - int... and Wraper class - Integer... are immutable
> AtomicDatatypes - AtomicInteger... is mutable

AtomicInteger as counter:

- `addAndGet()` – Atomically adds the given value to the current value and returns new value *after* the addition.
- `getAndAdd()` – Atomically adds the given value to the current value and returns old value.
- `incrementAndGet()` – Atomically increments the current value by 1 and returns new value *after* the increment. It is equivalent to **++i** operation.

- `getAndIncrement()` – Atomically increment the current value and returns old value. It is equivalent to **i++** operation.
- `decrementAndGet()` – Atomically decrements the current value by 1 and returns new value *after* the decrement. It is equivalent to **i- –** operation.
- `getAndDecrement()` – Atomically decrements the current value and returns old value. It is equivalent to **– -i** operation.

▼ Code

```
package com.java.concurrency;

import java.util.concurrent.atomic.AtomicInteger;

public class AtomicIntegerBasics
{
    public static void main(String[] args)
    {
        try
        {
            AtomicInteger atomicInteger = new AtomicInteger();

            System.out.println("atomicInteger: "+atomicInteger.get());        //initial 0

            System.out.println("atomicInteger.addAndGet(10): "+atomicInteger.addAndGet(10));

            AtomicInteger atomicInteger1 = new AtomicInteger(10);

            System.out.println("atomicInteger1: 10");

            System.out.println("atomicInteger.equals(atomicInteger1): "+atomicInteger.equals(atomicInteger1));
            //this is why we can't use it in place of integer

            System.out.println("atomicInteger.compareAndSet(10,20): "+atomicInteger.compareAndSet(10, 20));

            System.out.println("atomicInteger: "+atomicInteger.get());

            System.out.println("atomicInteger1.compareAndSet(12,30): "+atomicInteger1.compareAndSet(12,30));

            System.out.println("atomicInteger1: "+atomicInteger1.get());

            atomicInteger.set(22);

            System.out.println("atomicInteger.set(22): "+atomicInteger.get());
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}

//output
atomicInteger: 0
atomicInteger.addAndGet(10): 10
atomicInteger1: 10
atomicInteger.equals(atomicInteger1): false
atomicInteger.compareAndSet(10,20): true
atomicInteger: 20
atomicInteger1.compareAndSet(12,30): false
atomicInteger1: 10
atomicInteger.set(22): 22
```

▼ AtomicLong

See Program: `AtomicLongBasics.java`

```
package com.java.concurrency;

import java.util.concurrent.atomic.AtomicLong;

import java.util.function.Function;

public class AtomicLongBasics
{

    public static void main(String[] args)
    {
        try
        {
            AtomicLong atomicLong = new AtomicLong();       //will be common in both the functions

            Function<Long, Long> myLambda = (input) ->      //using lambda
            {
                long noOfCalls = atomicLong.incrementAndGet();

                System.out.println("Lambda called " + noOfCalls + " times.");

                return input * 2;
            };

            Function<Long, Long> newLambda = new Function<Long, Long>()
            {                                           //using anonymous class
                @Override
                public Long apply(Long input)
                {
                    long noOfCalls = atomicLong.incrementAndGet();

                    System.out.println("Lambda called " + noOfCalls + " times.");

                    return input * 2;
                }
            };

            System.out.println(myLambda.apply(1L));

            System.out.println(myLambda.apply(3L));

            System.out.println(myLambda.apply(5L));

            System.out.println(newLambda.apply(2L));

            System.out.println(newLambda.apply(4L));

            System.out.println(newLambda.apply(8L));
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
```

▼ Synchronized and Concurrenct hashmap

ConcurrentHashMap allows performing concurrent read and write operation. Hence, performance is relatively better than the Synchronized Map. In Synchronized HashMap, multiple threads can not access the map concurrently. Hence, the performance is relatively less than the ConcurrentHashMap.