Concurrency

An instance of Thread is just…an object. Like any other object in Java, it has variables and methods, and lives and dies on the heap. But a thread of execution is an individual process (a "lightweight" process) that has its own call stack. In Java, there is one thread per call stack—or, to think of it in reverse, one call stack per thread. Even if you don't create any new threads in your program, threads are back there running. The main() method, that starts the whole ball rolling, runs in one thread, called (surprisingly) the main thread.

The overloaded run(String s) method will be ignored by the Thread class unless you call it yourself. The Thread class expects a run() method with no arguments, and it will execute this method for you in a separate call stack after the thread has been started. With a run(String s) method, the Thread class won't call the method for you

common way to think about this is that the Thread is the"worker," and the Runnable is the "job"

Why wait() ,notify() and notifyAll() methods are in Object class instead of Thread class?

These methods works on the locks and locks are associated with Object and not Threads. Hence, it is in Object class.The methods wait(), notify() and notifyAll() are not only just methods, these are synchronization utility and used in communication mechanism among threads in Java. In java to achieve mutually exclusive access on objects, a threads needs to acquire lock and other threads needs to wait to acquire lock. And they don’t know which threads holds lock instead they just know the lock is hold by some other thread and they should wait for lock instead of knowing which thread is inside the synchronized block and asking them to release lock

Multithreading Benefits

Better resource utilization.
Simpler program design in some situations.
More responsive programs.

Volatile keyword

- The value of this variable will never be cached thread-locally: all reads and writes will go straight to "main memory";
- Access to the variable acts as though it is enclosed in a synchronized block, synchronized on itself.

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.

What is ThreadLocal? A simple example

As its name suggests, a single instance of ThreadLocal can store different values for each thread independently. Therefore, the value stored in a ThreadLocal instance is specific (local) to the current running Thread, any other code logic running on the same thread will see the same value, but not the values set on the same instance by other threads. (There are exceptions, like InhertiableThreadLocal, which inherits parent thread's values by default.) Let's consider this example:

We have a TransactionManager class that provide static methods to:

- Start a transaction with a generated ID
- Store that ID as a static field and provide a transaction ID getter method to other code logic that needs to know the current transaction ID.

```
public class TransactionManager {
   private static final ThreadLocal<String> context = new ThreadLocal<String>();

   public static void startTransaction() {
      //logic to start a transaction
      //...
      context.set(generatedId);
   }

   public static String getTransactionId() {
      return context.get();
   }

   public static void endTransaction() {
      //logic to end a transaction
      //…
      context.remove();
   }
}
```

Different thread that starts transactions via TransactionManager will get its own transaction ID stored in the context. Any logic within the same thread can call getTransactionId() later on to retrieve the value belongs/local to that Thread. So problem's solved!

Interrupts

An interrupt is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson.A thread sends an interrupt by invoking interrupt on the Thread object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must

support its own interruption.

Joins : The join method allows one thread to wait for the completion of another. If t is a Thread object whose thread is currently executing.

Thread Interference : Interference happens when two operations, running in different threads, but acting on the same data, interleave. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

Memory Consistency Errors : occurs when different threads have inconsistent views of the shared data.
Avoiding Memory Consistency Errors: This can be avoided by establishing a happens-before relationship . This relationship guarantees that memory write by one thread is visible to a read by other thread if the write operation happens-before the read operation. There are various actions that can form a happens-before relationship.
Synchronization , volatile keyword

There is several rules that we must keep in mind when using locks :

- Every mutable field shared between multiple threads must be guarded with a lock or made volatile, if you only need visibility
- Synchronize only the operations that must synchronized, this improves performance. But don't synchronize too few operations. Try to keep the lock only for short operations.
- Always know which locks are acquired and when there are acquired and by which thread
- An immutable object is always thread safe

Synchronization:- only a single thread can execute a block of code at the same time
Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: thread interference and memory consistency errors. The tool needed to prevent these errors is synchronization.
One Major **disadvantage of Java synchronized keyword** is that it doesn't allow concurrent read, which can potentially limit scalability
an ***intrinsic lock*** is implied by each use of the synchronized keyword. In this case, the locking is performed by Java behind the scenes. (This is distinct from the programmer using or defining an explicit lock object themselves.)
Each use of the synchronized keyword is associated with one of the two types of intrinsic lock:

- an "instance lock", attached to a single object
- a "static lock", attached to a class

The synchronized keyword can be used to mark four different types of blocks:

1. Instance methods
2. Static methods
3. Code blocks inside instance methods

4. Code blocks inside static methods

```
public synchronized void add(long value){
  this.count += value;
}
public void add(int value){
        synchronized(this){
                this.count += value;
        }
}
public static synchronized void log1(String msg1, String msg2){
  log.writeln(msg1);
  log.writeln(msg2);
}
public static void log2(String msg1, String msg2){
  synchronized(MyClass.class){
          log.writeln(msg1);
          log.writeln(msg2);
  }
}
```

Main Differences Between Locks and Synchronized Blocks

The main differences between a Lock and a synchronized block are:

- A synchronized block makes no guarantees about the sequence in which threads waiting to entering it are granted access.
- You cannot pass any parameters to the entry of a synchronized block. Thus, having a timeout trying to get access to a synchronized block is not possible.
- The synchronized block must be fully contained within a single method. A Lock can have it's calls to lock() and unlock() in separate methods.

Lock Methods

The Lock interface has the following primary methods:

- lock()
- lockInterruptibly()
- tryLock()
- tryLock(long timeout, TimeUnit timeUnit)
- unlock()

The lock() method locks the Lock instance if possible. If the Lock instance is already locked, the thread calling lock() is blocked until the Lock is unlocked.

The lockInterruptibly() method locks the Lock unless the thread calling the method has been interrupted. Additionally, if a thread is blocked waiting to lock the Lock via this method, and it is interrupted, it exits this method calls.

The tryLock() method attempts to lock the Lock instance immediately. It returns true if the locking succeeds, false if Lock is already locked. This method never blocks.

The tryLock(long timeout, TimeUnit timeUnit) works like the tryLock() method, except it

waits up the given timeout before giving up trying to lock the Lock.

The unlock() method unlocks the Lock instance. Typically, a Lock implementation will only allow the thread that has locked the Lock to call this method. Other threads calling this method may result in an unchecked exception (RuntimeException).

java.util.concurrent.locks

Interface Lock
All Known Implementing Classes:
ReentrantLock, ReentrantReadWriteLock ,ReadLock, WriteLock
ReentrantReadWriteLock.

Intrinsic lock / Monitor lock.
Intrinsic lock or monitor lock plays an important role in different aspects of synchronization :

- It enforces exclusive access to object's state .
- It establishes *Happens-before* relationship that is essential for visibility .

What are Reentrant Locks/Reentrant Synchronization
public class ReentrantLock extends Object implements Lock,Serializable
Reentrant Lock is a concrete implementation of Lock interface provided in Java concurrency package.ReentrantLock is mutual exclusive lock, similar to implicit locking provided by synchronized keyword in Java, with extended feature like fairness.Using optional “fairness” parameter with ReentrantLock. ReentrantLock accepts an optional “fairness” parameter in it’s constructor. Normally what happens is, whenever a thread releases the lock anyone of the waiting threads will get the chance to acquire that lock. But there is no predefined order or priority in the selection of the thread (at least from a programmers perspective).But if we are specifying the fairness parameter as “true” while creating a new ReentrantLock object, it gives us the guaranty that the longest waiting thread will get the lock next. Sounds pretty nice right?

This can be considered as a replacement for the traditional “wait-notify” method. The basic concept is, every thread need to acquire the lock before entering in to the critical section and should release it after finishing it. And its the most basic concept of synchronization.

ReentrantLock eliminates the use of “synchronized” keyword.

```
import java.util.concurrent.locks.ReentrantLock;
final ReentrantLock _lock = new ReentrantLock();
private void method() throws InterruptedException
{
  //Trying to enter the critical section
  _lock.lock(); // will wait until this thread gets the lock
  try
  {
    // critical section
  }
  finally
  {
    //releasing the lock so that other threads can get notifies
```

```
    _lock.unlock();
  }

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

2) Second difference between synchronized and Reentrant lock is tryLock() method. ReentrantLock provides convenient tryLock() method, which acquires lock only if its available or not held by any other thread. This reduce blocking of thread waiting for lock in Java application.

3) One more worth noting difference between ReentrantLock and synchronized keyword in Java is, ability to interrupt Thread while waiting for Lock. In case of synchronized keyword, a thread can be blocked waiting for lock, for an indefinite period of time and there was no way to control that. ReentrantLock provides a method called lockInterruptibly(), which can be used to interrupt thread when it is waiting for lock. Similarly tryLock() with timeout can be used to timeout if lock is not available in certain time period.

4) ReentrantLock also provides convenient method to get List of all threads waiting for lock.

Fairness:-Normally what happens is, whenever a thread releases the lock anyone of the waiting threads will get the chance to acquire that lock. But there is no predefined order or priority in the selection of the thread (at least from a programmers perspective).But if we are specifying the fairness parameter as “true” while creating a new ReentrantLock object, it gives us the guaranty that the longest waiting thread will get the lock next. Sounds pretty nice right

Atomic Access (Atomic Variable): In programming, an atomic action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete. java.util.concurrent package provide atomic methods that do not rely on synchronization.The atomic compareAndSet(CAS) method also has these memory consistency features,

Liveness : A concurrent application's ability to execute in a timely manner is known as its liveness. This section describes the most common kind of liveness problem, deadlock, and goes on to briefly describe two other liveness problems, starvation and livelock.

Starvation : Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Livelock: A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then livelock may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are

not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...

Immutable Objects(A Strategy for Defining Immutable Objects)

An object is considered immutable if its state cannot change after it is constructed.
The following rules define a simple strategy for creating immutable objects. Not all classes documented as "immutable" follow these rules. This does not necessarily mean the creators of these classes were sloppy — they may have good reason for believing that instances of their classes never change after construction. However, such strategies require sophisticated analysis and are not for beginners.

1. Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
2. Make all fields final and private.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
    - Don't provide methods that modify the mutable objects.
    - Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

Executors

In large-scale applications,allocating and deallocating many thread objects creates a significant memory management overhead. it makes sense to separate thread management. Thread pools manage a pool of worker threads. The thread pools contains a work queue which holds tasks waiting to get executed.
A thread pool can be described as a collection of Runnable objects (work queue) and a connections of running threads. These threads are constantly running and are checking the work query for new work. If there is new work to be done they execute this Runnable. The Thread class itself provides a method, e.g. execute(Runnable r) to add a new Runnable object to the work queue.
A thread pool is represented by an instance of the class ExecutorService. With an ExecutorService, you can submit task that will be completed in the future. Here are the type of thread pools you can create with the Executors class :

Single Thread Executor : A thread pool with only one thread. So all the submitted tasks will be executed sequentially. Method : Executors.newSingleThreadExecutor()

Cached Thread Pool : A thread pool that creates as many threads it needs to execute the task in parrallel. The old available threads will be reused for the new tasks. If a thread is not used during 60 seconds, it will be terminated and removed from the pool. Method : Executors.newCachedThreadPool()

Fixed Thread Pool : A thread pool with a fixed number of threads. If a thread is not available for the task, the task is put in queue waiting for an other task to ends. Method : Executors.newFixedThreadPool().

Scheduled Thread Pool : A thread pool made to schedule future task. Method : Executors.newScheduledThreadPool()

Single Thread Scheduled Pool : A thread pool with only one thread to schedule future task. Method : Executors.newSingleThreadScheduledExecutor()

Objects that encapsulate these functions are known as executors. The following subsections describe executors in detail.

Executor : Interfaces define the three executor object types.
Thread Pools : are the most common kind of executor implementation.
Fork/Join : is a framework (new in JDK 7) for taking advantage of multiple processors.

java.util.concurrent package defines three executor interfaces:-

Executor, a simple interface that supports launching new tasks.
ExecutorService, a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
ScheduledExecutorService, a subinterface of ExecutorService, supports future and/or periodic execution of tasks.

The Executor Interface:-- provides a single method, execute, designed to be a drop-in replacement for a common thread-creation idiom. If r is a Runnable object, and e is an Executor object you can replace
(new Thread(r)).start();
with
e.execute(r);

The ExecutorService interface supplements execute with a similar, but more versatile submit method. Like execute, submit accepts Runnable objects, but also accepts Callable objects, which allow the task to return a value. The submit method returns a Future object, which is used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks.
ExecutorService also provides methods for submitting large collections of Callable objects. Finally, ExecutorService provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should handle interrupts correctly.

The ScheduledExecutorService interface supplements the methods of its parent ExecutorService with schedule, which executes a Runnable or Callable task after a specified delay.
ExecutorService :There are a few different ways to delegate tasks for execution to an ExecutorService:
execute(Runnable)

submit(Runnable)
submit(Callable)
invokeAny(...)
invokeAll(...)
ExecutorService Shutdown:-When you are done using the ExecutorService you should shut it down, so the threads do not keep running.

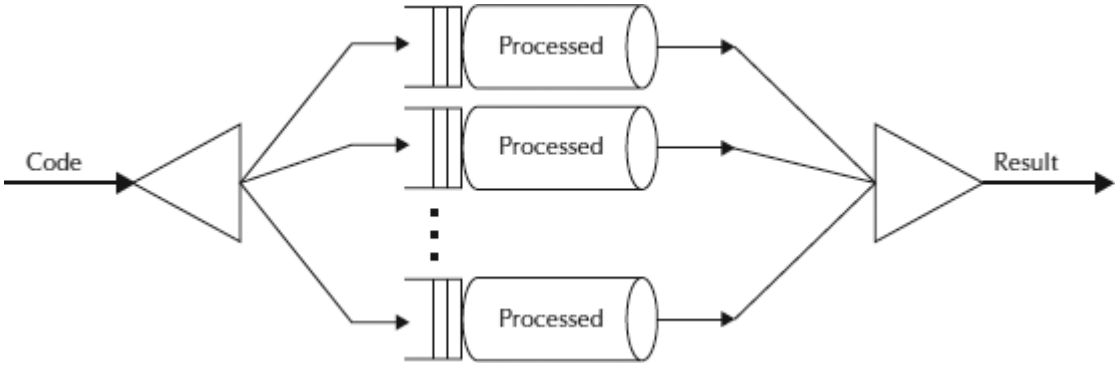
Futures and Callables :- Runnable do not return result. But if you submit a Callable object to an Executor the framework returns an object of type java.util.concurrent.Future. This Future object can be used to check the status of a Callable and to retrieve the result from the Callable.

Fork/Join

The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

As with any ExecutorService implementation, the fork/join framework distributes tasks to worker threads in a thread pool. The fork/join framework is distinct because it uses a work-stealing algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

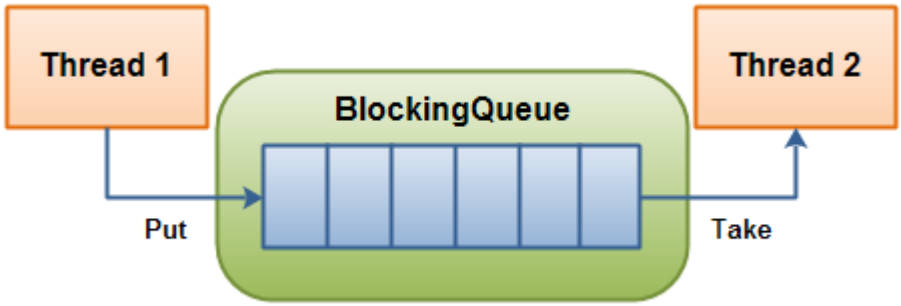
The center of the fork/join framework is the ForkJoinPool class, an extension of the AbstractExecutorService class. ForkJoinPool implements the core work-stealing algorithm and can execute ForkJoinTask processes.
Basically the **Fork-Join breaks the task at hand into mini-tasks** until the mini-task is simple enough that it can be solved without further breakups. It's **like a divide-and-conquer algorithm**. One important concept to note in this framework is that **ideally no worker thread is idle**. They implement a **work-stealing algorithm** in that idle workers steal the work from those workers who are busy.



BlockingQueue :- defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue. The Java BlockingQueue interface in the java.util.concurrent class represents a queue which is thread safe to put into, and take instances from Queue.BlockingQueue is full you put()method of BlockingaQueue will block ,if Queue is empty and take() method of BlockingQueue will block if Queue is full. This property makes BlockingQueue an ideal choice for implementing Producer consumer design pattern

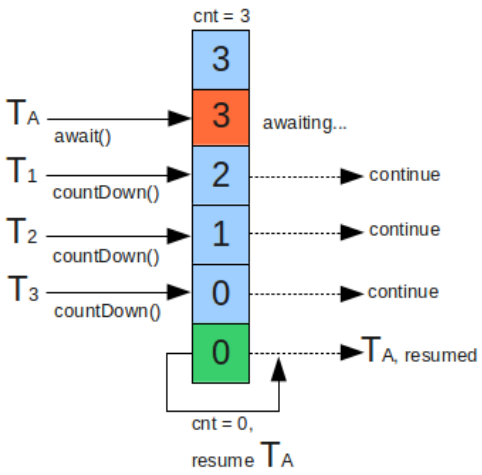
following implementations of the BlockingQueue interface

- ArrayBlockingQueue
- DelayQueue
- LinkedBlockingQueue
- PriorityBlockingQueue
- SynchronousQueue



CountDownLatch
A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.
A CountDownLatch is initialized with a given count. The await methods block until the current count reaches zero due to invocations of the countDown() method, after which all waiting threads are released and any subsequent invocations of await return immediately. This is a one-shot phenomenon -- the count cannot be reset. If you need a version that resets the count, consider using a CyclicBarrier.
Note that this is a **one-off process**: once the latch reaches zero, there is **no way to reset** the count.
In Java:

- the `CountDownLatch` object is constructed with the initial count;
- calling `countDown()` decrements the count by 1;
- the `await()` method will wait for the count to reach zero, or proceed immediately if the count *already* reached zero.



CyclicBarrier

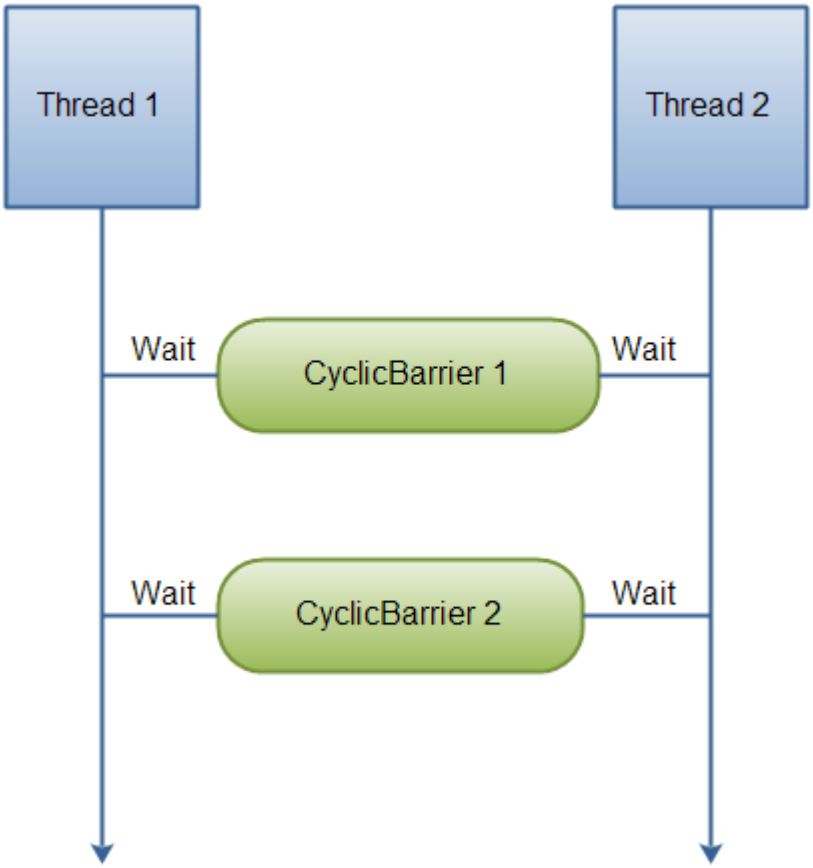
A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called cyclic because it can be re-used after the waiting threads are released.

A CyclicBarrier supports an optional Runnable command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This barrier action is useful for updating shared-state before any of the parties continue.
The CyclicBarrier is generally more useful than CountDownLatch in cases where:

a multithreaded operation occurs in stages or iterations, and;
a single-threaded operation is required between stages/iterations, for example, to combine the results of the previous multithreaded portion.

Overview of CyclicBarrier

Firstly, the barrier is constructed with the following:
the number of threads that will be participating in the parallel operation; optionally, an amalgamation routine to run at the end of each stage/iteration.
Then, at each stage (or on each iteration) of the operation:
each thread carries out its portion of the work;
after doing its portion of the work, each thread calls the barrier's await() method;
the await() method returns only when:
all threads have called await(); the amalgamation method has run (the barrier calls this on the last thread to call await() before releasing the awaiting threads).
if any of the threads is interrupted or times out while waiting for the barrier, then the barrier is "broken" and all other waiting threads receive a BrokenBarrierException.


Thread 1
Thread 2
Wait
CyclicBarrier 1
Wait
Wait
CyclicBarrier 2
Wait