**Thread**

1)using for multitasking at a time.

->Thread is a faster in the execution &lighter(consuming less memory).

2)when we are executing the java program then jave using 3-threads

->thread scheduler-allocates CPU time and memory for other threads.all threads should be registered with the thread scheduler to start the execution.

->Garbage Collector thread-clean the heap memory

->main thread-executes main method.

3)for creating thread class we have to inherit Thread class or implement Runnable interface.after that we have override run() method.task has to perform inside the run method.to start the Thread we have to use start() method on a Thread Object type.start() method registers the thread to the thread scheduler.once the thread is registered with thread scheduler then threads gets CPU time for execution.if thread is not registered with thread scheduler,such thread are not managed by thread scheduler.for wait sleep() method.

4)Runnable interface doesn’t contain start() method.we have to create thread object then we have to pass threadclass reference variable as a argument.

5)we can call run() method of a Thread class but then it will behave like a normal method. To actually execute it in a Thread, we need to start it using **Thread.start()** method.

**DIFF BETN RUNNABLE AND CALLABLE**

1) Runnable interface is older than Callable, there from JDK 1.0, while Callable is added on Java 5.0.

2) Runnable interface has run() method to define task while Callable interface uses call() method for task definition.

3) run() method does not return any value, it's return type is void while call method returns value.

4) Another difference on run and call method is that run method can not [throw](http://java67.blogspot.sg/2012/10/difference-between-throw-vs-throws-in.html) checked exception, while call method can throw checked exception in Java.

**public** Object call() **throws** Exception {

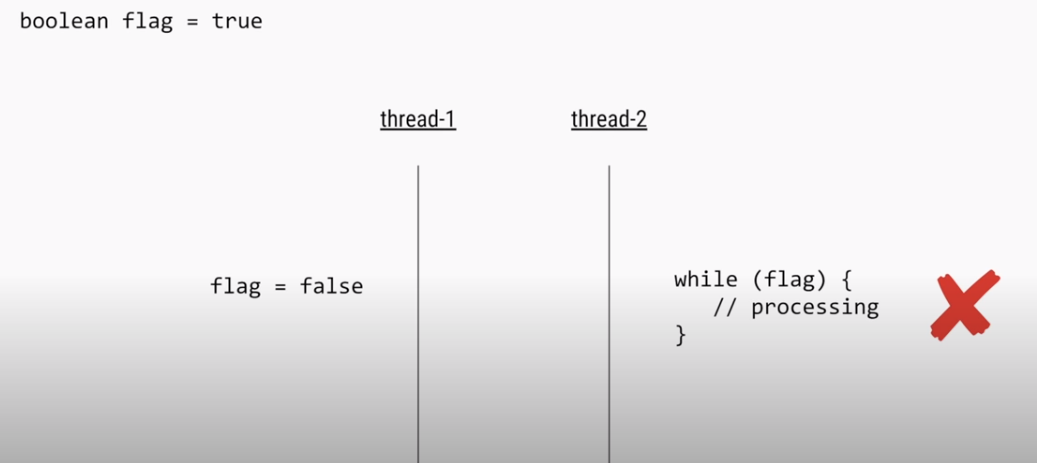
**returnnull**;

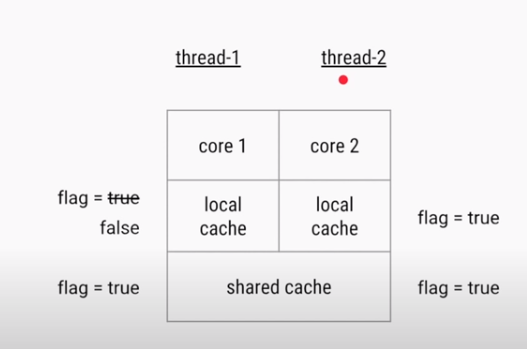
}

**volatile variable**

**->It is used to solve the visibility problems.**

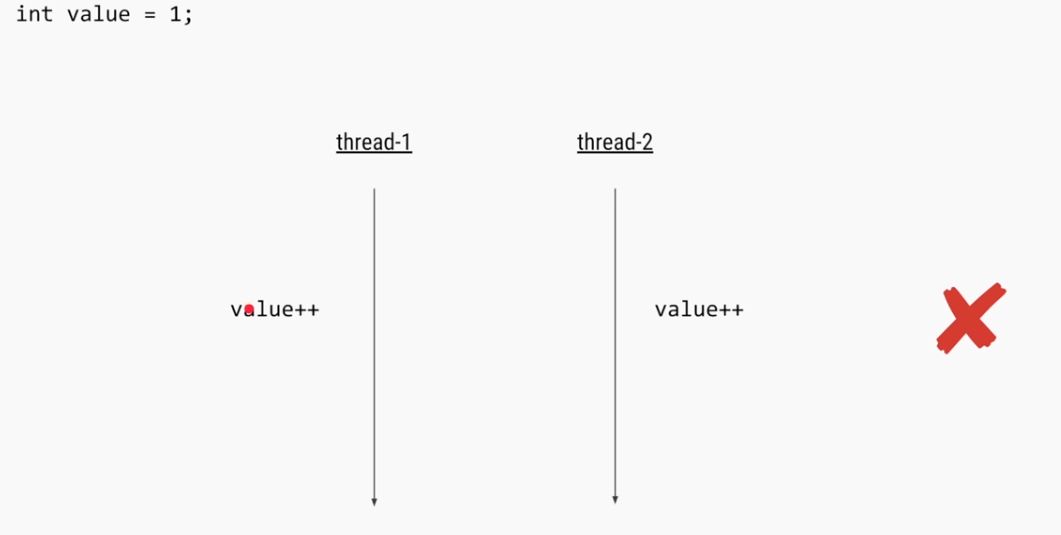
**->**if multiple thread is updating the variable then all threads will copy the variable in localcache and then will update therefore other threads doesn’t know the update value.therefore using volatile keyword threads will update the value in main memory then all threads will access the updated value.

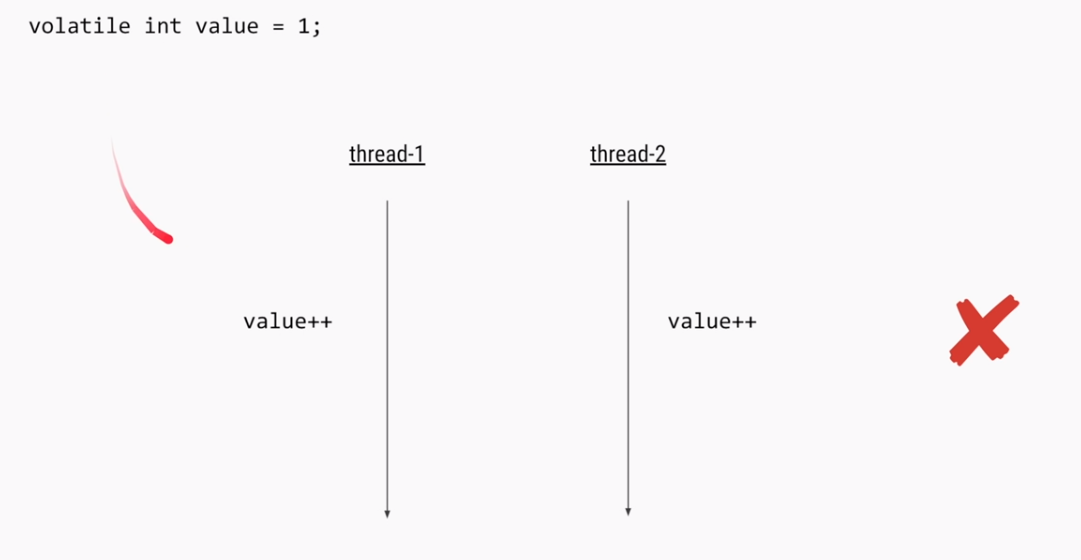


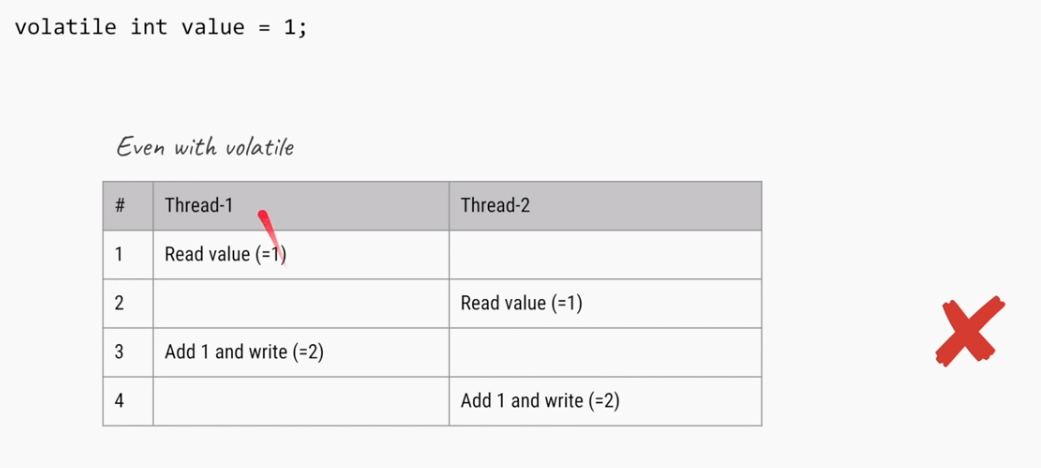


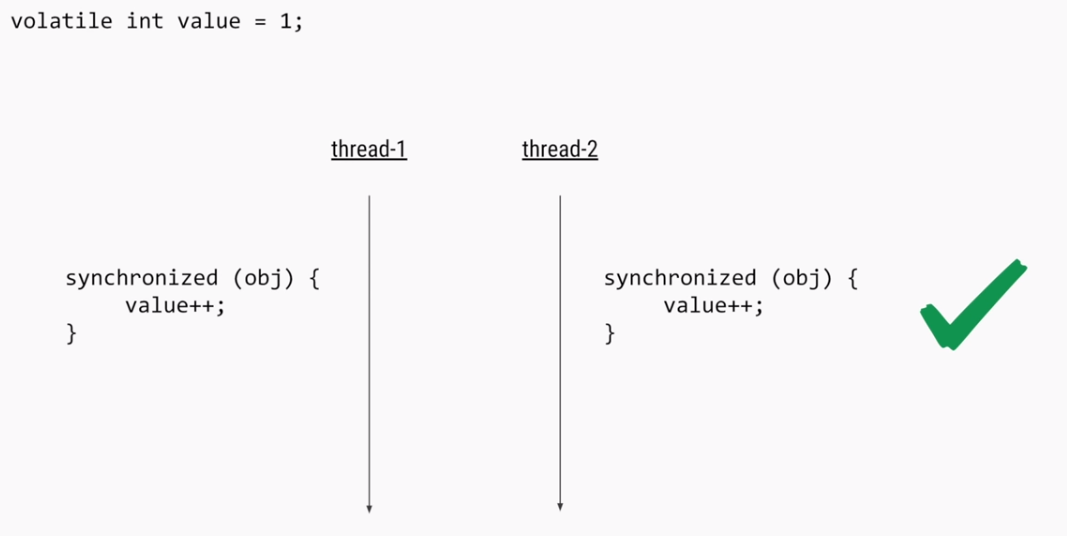
**atomic variable**

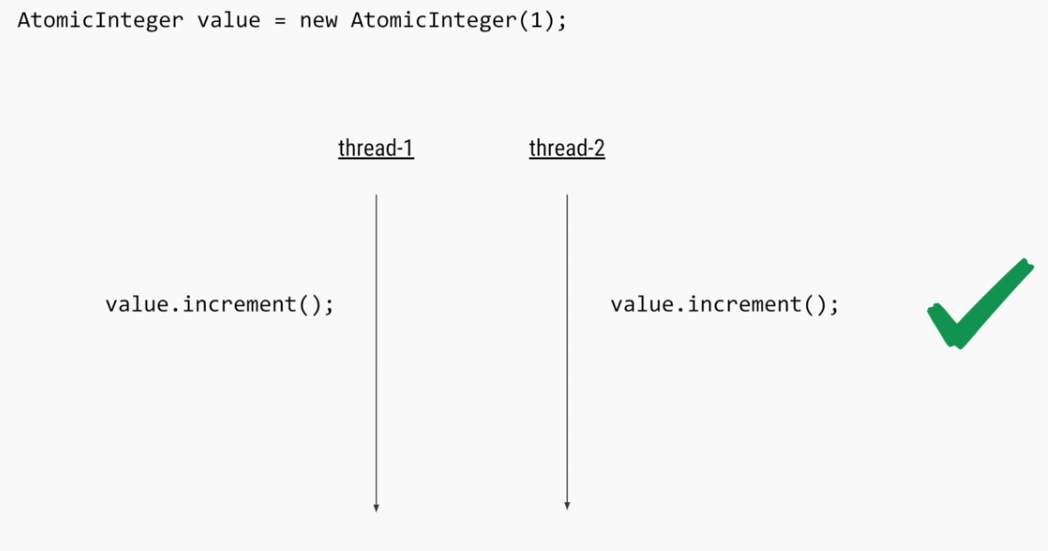
* Suppose we have to compound operation(read and write) atomically then use atomic variable.

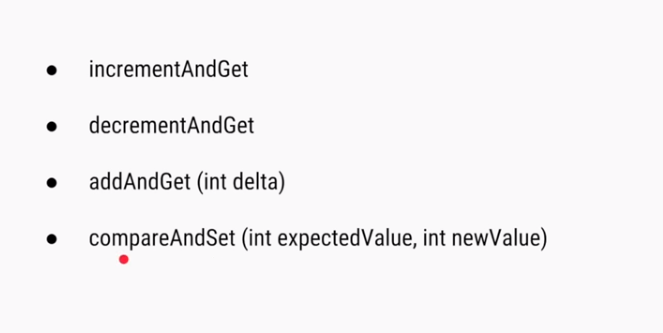
****

****

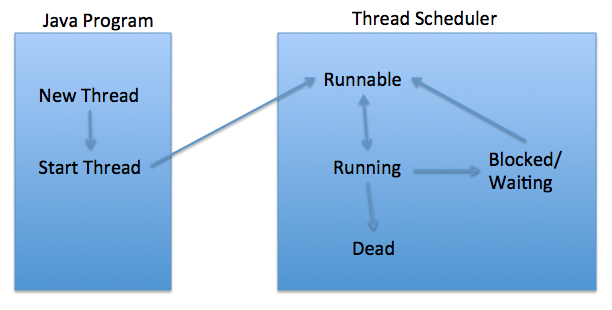
****

****

****

****

**State of Thread**



**New**

When we create a new Thread object using *new* operator, thread state is New Thread. At this point, thread is not alive and it’s a state internal to Java programming.

**Runnable**

When we call start() function on Thread object, it’s state is changed to Runnable. The control is given to Thread scheduler to finish it’s execution. Whether to run this thread instantly or keep it in runnable thread pool before running, depends on the OS implementation of thread scheduler.

**Running**

When thread is executing, it’s state is changed to Running. Thread scheduler picks one of the thread from the runnable thread pool and change it’s state to Running. Then CPU starts executing this thread. A thread can change state to Runnable, Dead or Blocked from running state depends on time slicing, thread completion of run() method or waiting for some resources.

**Blocked/Waiting**

A thread can be waiting for other thread to finish using [thread join](http://www.journaldev.com/1024/java-thread-join-example) or it can be waiting for some resources to available. For example [producer consumer problem](http://www.journaldev.com/1034/java-blockingqueue-example) or [waiter notifier implementation](http://www.journaldev.com/1037/java-thread-wait-notify-and-notifyall-example) or IO resources, then it’s state is changed to Waiting. Once the thread wait state is over, it’s state is changed to Runnable and it’s moved back to runnable thread pool.

**Dead**

Once the thread finished executing, it’s state is changed to Dead and it’s considered to be not alive.

**double checked locking of Singleton**

public class Singleton{

 private static volatile Singleton \_instance;

public static Singleton getInstance(){

**if(\_instance == null){**

 synchronized(Singleton.class){

**if(\_instance == null)**

\_instance = new Singleton();

} }

 return \_instance;

}

**Thread Synchronization**

1)when at a multiper user accesing the common database or information or shared information then data manipulation or data corruption will occur.to avoid this problem we can use synchronized keyword with common information.

2)then it will allow only one thread at a time.

3)in synchronization,when a thread accessing the non-static synchronized resource block then it will lock the object.execute all the statement then it returns control to other thread.thatsy other thread can’t access any nonstaticresource.but we can access static resource.

4)when thread enters into static synchronized block then it will lock to class member then we can’t access the any static resource.

5)both object lock & class lock can exist.

## Race Conditions

1)Race conditions occurs when two thread operate on same object without proper synchronization.  
  
2)Classical example of Race condition is incrementing a counter since increment is not an atomic operation.  
  
  
**publicclassAtomicCounter**{

**privatefinal**AtomicInteger value =**new**AtomicInteger(0);

**public**intgetValue(){

**return**value.get();

}

**public**intgetNextValue(){

**return**value.incrementAndGet();

}

**public**intgetPreviousValue(){

**return**value.decrementAndGet();

}

}

How to stop the thread

1)Earlier there was a stop method exists in Thread Class but Java deprecated that method citing some safety reason.

2)By default a Thread stops when execution of run() method finish either normally or due to any Exception.

3)In this article we will How to Stop Thread in Java by using a boolean State variable or flag. Using flag to stop Thread is very popular way  of stopping thread and its also safe, because it doesn't do anything special rather than helping run() method to finish it self.

publicclassTaskimplementsRunnable {

privatevolatilebooleanisRunning = true;

publicvoid run() {

while (**isRunning**) {

//do work

}

}

publicvoid kill() {

isRunning = false;

}

}

To stop it just call

task.kill();

**Diff between yield(), join() and sleep**

1. yield() method pauses the currently executing thread temporarily for giving a chance to the remaining waiting threads of the same priority to execute. If there is no waiting thread or all the waiting threads have a lower priority then the same thread will continue its execution.
2. join() If any executing thread t1 calls join() on t2 i.e; t2.join() immediately t1 will enter into waiting state until t2 completes its execution.
3. sleep() Based on our requirement we can make a thread to be in sleeping state for a specified period of time.

**Thread Deadlock**

1)when a thread enters into resource block then it locks the object and wait for the updates from other resource block of the same class or by other threads.the other threads can not updates waiting thread until thread releases the lock.the waiting thread will not release the lock until the other thread updates.this is known as deadlock.

2)thread deadlock can be avoided by using inter-thread communication.the inter thread communication can achived by following methods.

->wait()-when a thread executes wait() method,the thread releases the lock and enters into the waiting pool until it gets modification from other threads.

->notify()-whenever other thread notifies the thread in the wait pool then waiting thread resupes their action.notify() method notifies only one thread at a time.

->notifyAll()–notifyAll() method notifies all the threads in the wait pool based on JVM implementation,one thread resumes at a time.

3)wait(),notify() and notifyAll() methods should be always used inside the synchronized block.

4)whenever a thread is created then thread properties like thread name,id& priority will be set.

->name-setName(),getName()

->id-getID()

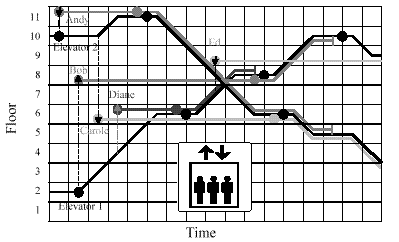
->priority-setPriority(),getPriority(),that will fix 1,2,3,4,5,…..10.avg priority will be 5.

|  |  |
| --- | --- |
| Multitasking | Multithreading |
| 1)tasks are known as heavy weight process.  2)heavy-weight process require separate address space.  3)allows diff. program to run concurrently on the same system.e.g.running java compiler while using the text editor.  4)interprocess communication is expensive & limited.  5)context switching from one process to another is also costly. | 1)light weight process  2)share the same address space.  3)allows diff parts of the same program to run concurrently.e.g a word processor that is printing and formatting text at the same time.  4)interthread communication is inexpensive.  5)context switching from one thread to another thread is low cost. |

**Concurrency**

**Example of a Concurrent, Real-time System: An Elevator System**

As an example to illustrate the concepts to be discussed, we will use an elevator system. More precisely, we mean a computer system designed to control a group of elevators at one location in a building. Obviously there may be many things going on concurrently within a group of elevators—or nothing at all! At any point in time someone on any floor may request an elevator, and other requests may be pending. Some of the elevators may be idle, while others are either carrying passengers, or going to answer a call, or both. Doors must open and close at appropriate times. Passengers may be obstructing the doors, or pressing door open or close buttons, or selecting floors—then changing their minds. Displays need to be updated, motors need to be controlled, and so on, all under the supervision of the elevator control system. Overall, it’s a good model for exploring concurrency concepts, and one for which we share a reasonably common degree of understanding and a working vocabulary.



As potential passengers place demands upon the system at different times, the system attempts to provide the best overall service by selecting elevators to answer calls based upon their current states and projected response times. For example, when the first potential passenger, Andy, calls for an elevator to go down, both are idle, so the closest one, Elevator 2, responds, although it must first travel upward to get to Andy. On the other hand, a few moments later when the second potential passenger, Bob, requests an elevator to go up, the more distant Elevator 1 responds, since it is known that Elevator 2 must travel downward to an as-yet-unknown destination before it can answer an up call from below.

In **asynchronous communication** the sending activity forwards its information regardless of whether the receiver is ready to receive it or not.

**Synchronous communication** includes synchronization between the sender and the receiver in addition to the exchange of information.If one activity (sender or receiver) is ready to communicate before the other, it will be suspended until the other one becomes ready as well. For this reason, this mode of communication is sometimes referred to as *rendezvous*.

Keep in mind that true concurrency of processes or threads is only possible on multipocessors with concurrent execution of processes or threads;

## Optimistic and Pessimistic Locking

1. Traditional locking mechanisms, e.g. using *synchronized* keyword in java, is said to be pessimistic technique of locking or multi-threading. It asks you to first guarantee that no other thread will interfere in between certain operation (i.e. lock the object), and then only allow you access to any instance/method.

It’s much like saying “please close the door first; otherwise some other crook will come in and rearrange your stuff”.

1. Though above approach is safe and it does work, but it put a significant penalty on your application in terms of performance. Reason is simple that waiting threads can not do anything unless they also get a chance and perform the guarded operation.
2. There exist one more approach which is more efficient in performance, and it optimistic in nature. In this approach, you proceed with an update, being hopeful that you can complete it without interference.
3. **Compare and Swap** is a good example of such optimistic approach.Atomic operations supported in classes(AtomicInteger, AtomicLong) such as AtomicInteger, AtomicLongetc.These classes internally rely on an algorithm named CAS (compare and swap).
4. There are 3 parameters for a CAS operation:

* A memory location V where value has to be replaced
* Old value A which was read by thread last time
* New value B which should be written over V

1. Let’s break the whole CAS operation in steps:

**1) Thread 1 and 2 want to increment it, they both read the value and increment it to 11.**

*V = 10, A = 0, B = 0*

**2) Now thread 1 comes first and compare V with it’s last read value:**

*V = 10, A = 10, B = 11*

if     A = V

  V = B

else

operation failed

return V

Clearly the value of V will be overwritten as 11, i.e. operation was successful.

**3) Thread 2 comes and try the same operation as thread 1**

*V = 11, A = 10, B = 11*

if     A = V

  V = B

else

operation failed

return V

**4) In this case, V is not equal to A, so value is not replaced and current value of V i.e. 11 is returned. Now thread 2, again retry this operation with values:**

*V = 11, A = 11, B = 12*

And this time, condition is met and incremented value 12 is returned to thread 2.

1. In summary, when multiple threads attempt to update the same variable simultaneously using CAS, one wins and updates the variable’s value, and the rest lose. But the losers are not punished by suspension of thread. They are free to retry the operation or simply do nothing.

**Adavantages of Executor over Traditional Thread**

1. **Developer have to manage thread management stuff but using executor no need to worry.**
2. **Always good approach is reusing the resource rather creating resources.**
3. **Also creating the resources is time consuming.**
4. **When we keep on creating the resources then might be we get the exception of stackoverflow and JVM will crash.**

**Java Executor Framework**

**ExecutorService**

1)Creating thread is expensive in terms of time and resource. If you create thread at time of request it will slow down your response time, also there is only a limited number of threads a process can create. To avoid both of these issue, a pool of thread is created .

2) Java API provides Executor framework, which allows you to create different types of thread pools e.g. single thread pool, which process one task at a time, fixed thread pool (a pool of fixed number of thread) or cached thread pool (an expandable thread pool suitable for applications with many short lived tasks).

**Fixed Thread Pool Executor**

1. Used to improved performance and better system resource utilization by limiting the maximum number of threads in thread pool.

**Cached Thread Pool Executor**

Used to create the thread on demand and all the task will execute concurrently.

**Single Thread Executor**

Used to run the task sequentially. If multiple threads are using a common resource then we should use single thread executor.

**ScheduledThreadPoolExecutor**

1. Suppose we want to execute a task after a period of time or to execute a task periodically or schedule the task then we can use [ScheduledThreadPoolExecutor](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html) class.
2. To execute a task in this scheduled executor after a period of time, you have to use the schedule() method. This method receives the following three parameters:

* The task you want to execute
* The period of time you want the task to wait before its execution
* The unit of the period of time, specified as a constant of the TimeUnit class

1. Execute a task periodically with scheduleAtFixedRate() method.This method accepts four parameters:

* the task you want to execute periodically,
* the delay of time until the first execution of the task,
* the period between two executions,
* and the time unit of the second and third parameters.

**ThreadPoolExecutor + Callable + Future**

Suppose we want to return a result after processing the task. Then we can use the Java Concurrency API with interfaces [Callable](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html) and [Future](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html).

**Throttling Task Submission Rate with ThreadPoolExecutor and Semaphore**

1. Suppose we want to configure the maximum number of concurrent connections to the web-server.If connections will exceed than the maximum number then they have to wait until some other connections are freed or closed. This limitation can be taken as throttling.
2. Throttling will help in keeping the number of tasks in queue in limit so that no task get rejected. It essentially removes the necessity of RejectedExecutionHandler as well.
3. CustomThreadPoolExecutor which had following capabilities:

* Tasks being submitted to blocking queue
* An executor which picks up the task from queue and execute them
* Had overridden **beforeExecute()** and **afterExecute()** methods to perform some extra activities if needed
* Attached a RejectedExecutionHandler which handle the task if it got rejected because the queue was full.

1. In this solution, we will create a [Semaphore](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html) with a number which must be equal to maximum number of tasks in blocking queue at any given point of time. So the approach works like this:

* Before executing a task a lock in semaphore is requested
* If lock is acquired then execution works normally; Otherwise retry will happen until lock is acquired
* Once task is completed; lock is released to semaphore

**BlockingQueue**

1. If you remember solving the [producer-consumer problem,](https://en.wikipedia.org/wiki/Producer-consumer_problem)  consumer had to wait until producer put something in resource queue. This problem can be easily solved using new BlockingQueue.
2. BlockingQueue works on following rules:

* If fewer than corePoolSize threads are running, the Executor always prefers adding a new thread rather than queuing.
* If corePoolSize or more threads are running, the Executor always prefers queuing a request rather than adding a new thread.
* If a request cannot be queued, a new thread is created unless this would exceed maximumPoolSize, in which case, the task will be rejected.

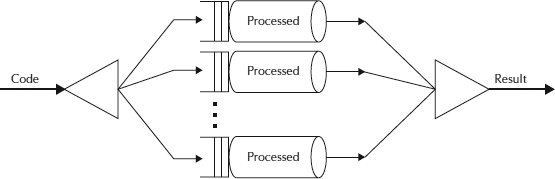
**UncaughtExceptionHandler**

1. Used to catch and treat the unchecked exceptions thrown in a Thread object to avoid the program ending.

#### Advance Concurrency

**Fork/Join Framework**

1. 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](http://en.wikipedia.org/wiki/Divide_and_conquer_algorithms).



Result solve(Problem problem) {

if (problem is small)

directly solve problem

else {

split problem into independent parts

fork new subtasks to solve each part

join all subtasks

compose result from subresults

}

}

**CountDownLatch**

1. This class **enables a java thread to wait until other set of threads completes** their tasks. e.g. Application’s main thread want to wait, till other service threads which are responsible for starting framework services have completed started all services.
2. CountDownLatch works by having a counter initialized with number of threads, which is decremented each time a thread complete its execution. When count reaches to zero, it means all threads have completed their execution, and thread waiting on latch resume the execution.
3. Pseudo code for CountDownLatch can be written like this:

//Main thread start

//Create CountDownLatch for N threads

//Create and start N threads

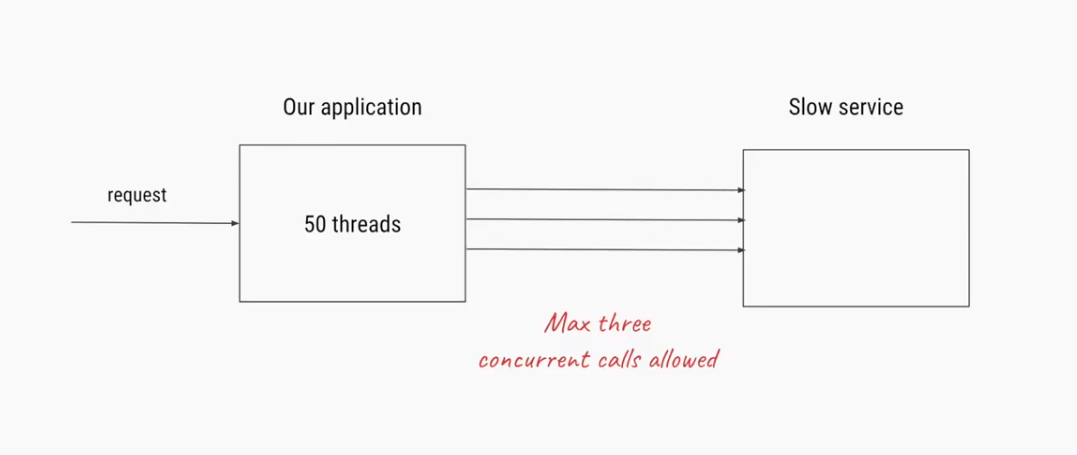
//Main thread wait on latch

//N threads completes there tasks are returns

//Main thread resume execution

**Semaphore**

1. Suppose we want to restrict the use of any resources in limited quantity then semaphore is the best choice
2. Ex: we have one slow service that’s allow only 3 thraeds at a time then we should use semaphore. It manage only 3 permits.
3. Methods: acquire(), tryAcquire(),tryAcquire(timeout), availablePermits(), new Semaphore(count,fairness)



**Binary Semaphore**

1. wheneverwe have a requirement for protecting the access to a SINGLE resource accessed by multiple threads, you can use Binary Semaphore.
2. To show the usage of binary semaphore, we are going to implement a print queue that can be used by concurrent tasks to print their jobs. This print queue will be protected by a binary semaphore, so only one thread can print at a time.

**Lock Interface**

1. A java.util.concurrent.locks.Lock is a thread synchronization mechanism just like synchronized blocks. A Lock is, however, more flexible and more sophisticated than a synchronized block. Since Lock is an interface, you need to use one of its implementations to use a Lock in your applications. ReentrantLock is one such implementation of Lock interface.
2. The main differences between a Lock and a synchronized block are:

* Having a timeout trying to get access to a synchronized block is not possible. Using [Lock.tryLock(long timeout, TimeUnittimeUnit)](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html" \l "tryLock%28long,%20java.util.concurrent.TimeUnit%29" \t "_blank" \o "lock method), it is possible.
* The synchronized block must be fully contained within a single method. A Lock can have it’s calls to lock() andunlock() in separate methods.

**Creating Threads Using java.util.concurrent.ThreadFactory**

1. we usually create threads using two ways i.e. [extending thread class and implementing runnable interface](http://howtodoinjava.com/core-java/multi-threading/difference-between-implements-runnable-and-extends-thread-in-java/). Java also provides an interface, the ThreadFactory interface, to create your own Thread object factory.

**ThreadLocal Variables**

1. use of thread local is when you have some object that is not thread-safe, but you want to avoid synchronizing access to that object using synchronized keyword/block. Instead, give each thread its own instance of the object to work with.
2. This class has following methods:

* get() : Returns the value in the current thread’s copy of this thread-local variable.
* initialValue() : Returns the current thread’s “initial value” for this thread-local variable.
* remove() : Removes the current thread’s value for this thread-local variable.
* set(T value) : Sets the current thread’s copy of this thread-local variable to the specified value.

**Java Inter-thread Communication using Piped Streams**

1. Pipe streams are just like real plumbing pipes.  You put things into to a pipe at one end – using some methods.  Then you receive the same things back from the pipe stream at the other end  – using some other methods. They come out in [FIFO order](http://en.wikipedia.org/wiki/FIFO), first-in first-out, just like from real plumbing pipes.
2. PipedReader is an extension of [Reader](http://docs.oracle.com/javase/7/docs/api/java/io/Reader.html) class which is used for reading character streams. Its read() method reads the connected PipedWriter’s stream. Similarly, PipedWriter is an extension of Writer class and does all the things which Reader class contracts.
3. A writer can be connected to a reader by following two methods:

* Using constructor PipedWriter(PipedReaderpr)
* Using connect(PipedReaderpr) method

1. Once connected through any of above ways, any thread can write data in stream using write(….) methods, and data will be available to reader and can be read using read() method.

#### Concurrent Collections

**ConcurrentHashMap**

1. The ConcurrentHashMap is very similar to the HashMap class, except that ConcurrentHashMap offers internally maintained concurrency. It means you do not need to have synchronized blocks when accessing ConcurrentHashMap in multithreaded application.

//Initialize ConcurrentHashMap instance

ConcurrentHashMap<String, Integer> m = newConcurrentHashMap<String, Integer>();

//Print all values stored in ConcurrentHashMap instance

foreach (Entry<String, Integer> e : m.entrySet())

{

system.out.println(e.getKey()+"="+e.getValue());

}

1. Above code yet provides thread safety, still it can decrease the performance of application.

To understand that we need to understand the internal working of ConcurrentHashMap class. And the best way to start is look at the constructor arguments. Fully parametrized constructor of ConcurrentHashMap takes 3 parameters, initialCapacity, loadFactor and concurrencyLevel.

1. A good approach can be having initialization like this:

ConcurrentHashMap<String, Integer> instance = newConcurrentHashMap<String, Integer>(16, 0.9f, 1);

**ConcurrentLinkedDeque**

1. Concurrent lists allow the various threads to add or remove elements in the list at a time without producing any data inconsistency. And non-blocking lists provide operations that, if the operation can’t be done immediately, lists throw an exception or return a null value, depending on the operation.
2. Java 7 has introduced the[ConcurrentLinkedDeque](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedDeque.html) class that implements a non-blocking concurrent list and in this tutorial, we will learn to use this class.

**Exceptions in Threads**

1. Leaking exceptions from thread cannot be caught directly.
2. For that we have to implement Thread.UncaughtExceptionHandler interface and override uncaughtException(Thread t, Throwable t) method.