Atomic Operations:

* Atomic operations can be accessed by one thread at a time.
* We can make operations on a variable atomic by using Atomic package of java.
* For example we can make a AtomicLong for making operations on the long variable as atomic.
* **This is used for making “Thread Safe State Variables”.**

Case of failure:

* This is going to fail in the scenario where there are multiple variables that are atomic variables
* The operations on each of those variables are atomic but not when taking the operations on both the variables together. Thus it’s going to cause race condition.

**In thread safety the “Invariants should be preserved regardless of timing or interleaving of operations in multiple threads”.**

Intrinsic Locks:

* To save us from these problems where we want that the invariants to be preserved, locks come into picture.
* We can lock a block of statements thus making them synchronized.
* It acts as mutexes.
* No thread executing a synchronized block can observer another thread. **It’s like an isolated world.**
* **a reference to an object will serve as a lock.**
* **static synchronized methods use Class object for the lock.**
* Every object has a built-in lock.
  + Locking doesn’t prevent other threads from accessing the object
  + The only thing it prevents in other threads from accessing the lock represented by the object.
* Problem with these locks:
  + When a thread acquire locks it blocks other thread which are waiting for the lock to get released.
* A very important property of Java locks.
  + These locks are **Reentrant**
    - This make the lock on the basis of the thread
    - When thread acquire the lock it can visit the same synchronized block again till he is having the acquired lock.
    - Thus this helps in preventing the dead lock in the scenarios where we are calling the super method of the child class and both the methods are synchronized.
* If synchronization is used to coordinate access to a variable, it is needed everywhere where the variable is access. Though I think ti’s predictable that this is going to happen.
* Each variable that participating in the invariant must be protected by the same lock
  + It means essentially that the invariant should be present in the same class (as objects in java are the locks)
  + So all the synchronized statements should be present in a single class.
* Narrowing the scope of synchronized block helps in improving concurrency
* Computation heavy block of could should be avoided in synchronized block.
  + As it would make the threads wait in queue.
* Acquiring and releasing a lock has some overhead so it’s undesirable to break down the synchronized block too far.

**Additional locking is always required when we are combining multiple statements into once statement. Even though each statements or variable are synchronized.**

**Sharing Objects:**

* **Visibility**
  + The we see in the program might not be the case when we are focusing on the multi threaded application.
  + A simple scenario is when we have two threads…one is main thread and another the new spawned thread.
    - There would be two variables readyToRead(boolean) and number(integer).
    - At the beginning readyToRead would be false and number = 0.
    - Our spawned thread would wait for the readyToRead become true and then would print the number.
  + Now lets take the execution of statements as follows:
    - new SpawnedThread().start().
    - number = 42;
    - readyToRead = true;
  + Looking at the statements order we can think that the spawnedThread would be waiting for the readToRead to become true.
  + When it becomes true it would print the number 42.
  + But in case of multiThread that might not be the case. Strange!!??
  + This would lead to stale value.
  + **Reason:** for the JVM to take full advantage of the multiprocessor hardware the compiler is permitted to reorder the operations and cache values in the registers and also permits the CPUs to reorder operations and cache’s values in the processor’s specific caches.
  + So it can be the case that the output printed is 0 instead of 42 because of reordering.
  + **Solution:** Is to use the synchronization rather than to believe what is visible!!
* Preventing stale data for a variable by setting both get and set as synchronized and guarding the variable by “this” object to attain the lock.
* **NonAtomic 64-bit operations:**
  + reading a variable without synchronization leads to stale value, that is not up-to date. Though it has been updated last by some other thread.
  + This out-of-thin-air safety enables u to see values placed by some other thread rather than some random value.
  + But this is not the case with the 64bit numerical variables(long and double) which are not declared **volatile**.
  + Reason: The JVM required the fetch and store operations to be **atomic**. But for non-volatile long and double values the JVM treat a 64-bit read and write as **two separate 32-bit operations**.
    - Thus if these separate 32-bit operations happen in two separate threads it is possible to get high 32 bit as different values and lower 32bits as another value, thus giving out some random value.
  + Thus it is not safe to use mutable long and double variables in multithreading unless they are volatile or guarded by lock.
* **Volatile Variables:**
  + A weaker form of synchronization to ensure the updates to a variable are propagated predictably.
  + The complier and runtime are put on a notice that the variable is shared and no operations on it should be reordered with other memory changes.
  + These are not caches in the registers or caches where they are hidden from other processors otherwise.
  + Read of a volatile variables returns the most recent write by any thread.
  + Yet, accessing a volatile variable doesn’t perform any locking and so cannot cause any the executing thread to block.
  + Volatile reads are just a little more expensive than non-volatile reads on current processor architecture.
  + Suppose a thread A has some variables visible before writing to a volatile variable x and a thread B reads the volatile variable x.
    - The variables that were visible to A before writing to x are visible to B after B reads the volatile variable x.
    - Thus writing a volatile variables is like exiting from a synchronized block and reading the volatile variable is like entering in a synchronized block.
  + It makes the code a little difficult to understand that the locking mechanism.
  + Can use used as a status flag.
  + Volatile variables are not strong enough to make increment operation atomic (count++).
  + Another way is to use Atomic Variables which can be used as “better volatile variables"
  + Usage of the volatile variables:
    - write to the variable doesn’t depend on it’s current value or a single thread always updates the value.
    - it doesn’t participate in invariants with other state variables.
    - locking is not required for any other reason when the variable is being accessed.
* **Publication and Escape**
  + publishing an object means to make it available out of the current scope
    - like storing a reference to it where the other code can find it
    - returning it from a non-private method
    - passing to a method of another class.
  + publishing inner state of objects will sacrifice the encapsulation thus making it difficult to preserve the variant.
  + publishing an object when it should not have been means that the object has **escaped.**
  + publishing an object also publishes any reference other other objects that its’ holding.
  + publishing one object might indirectly publish other others.
    - Like we are publishing a list.
    - The other thread can read and change the contents of the list.
* **Thread Confinement:**
  + Thread confinement is confining the data to a thread only.
  + Sharing data, mutable data requires synchronization. One way to avoid the requirement is to avoid sharing any data among threads.
  + A simple solution is to make data visible to one single thread. Thus making the object that is shared to only one thread as thread safe.
  + Ad-hoc thread confinement
    - just do thread confinement by design of the program.
  + ***Stack confinement:***
    - use the local variables rather than global variables.
    - local variables are confined to the thread.
    - They exists on the thread stack.
    - If you are calling a method that has the variables initialized inside the method
      * The variables exists on the thread stack.
      * As there is no way to obtain a reference to a **primitive variable**, so that language semantics ensure that primitive local variables are always stack confined.
    - In case of a reference to other object. For example a set.
      * Unless we give out the reference(publish it), the confinement won’t be violated.
    - **NOTE:** The problem here is, the developer might be writing the code thinking that the confinement of the variable inside a method would make it thread safe but the other developer might not know this and when they publish any of the reference the confinement is broken so it’s best to document this if we are doing stack confinement using local variables.
  + ***ThreadLocal***:
    - It’s like a map which hold the values of an object separately for each thread.
    - It provides get and set method:
      * get gives the values that is being set by the current executing thread.
    - Where can we use it
      * consider a scenario where we require a temporary buffer and wants to avoid reallocation of memory for the shared buffer among threads every time a new thread is accessing it. So it’s better to create a temporary buffer for each thread rather than doing reallocation everytime.
    - The thread specific values are stored in the thread itself. When the thread terminates, the thread specific values can be garbage collected.
* **Immutability*:***
  + immutable objects are inherently thread safe.
  + simply declaring a field final doesn’t make it immutable
    - since final fields can hold reference to mutable objects.
  + There is difference between the object being immutable and the reference to it being immutable.
  + As the immutable object can’t change.
    - But the immutable reference to a mutable object can still change the state of the object.
  + ***Final Fields***
    - they are initialization safe
    - it tells the maintainers that the fields are not expected to change.
    - Though the reference is immutable the object referenced can be changed. So always check for that.
  + ***Using Volatile to publish immutable objects:***
    - When ever a group of data is to handled atomically, it’s good to create an immutable class holder for them, with a mutable holder object.
    - The mutable holder can be volatile thus ensuring atomicity.
    - How it works?
      * We can create a wrapper class that has all final fields say this class is A
        + All fields of this class are final
        + There is just initialization and get
        + Get always give back the copy of required field(if it’s an array or other mutable type)
        + Thus we have made the object of A as immutable and whenever a thread does get it would be having a different copy of the required field value.
      * How we can create a volatile field of class A in our main class.
      * volatility provides synchronization.
      * Now this gives us synchronization because of volatile field and as we have made the class A as immutable we can use it safely across multiple thread. Each thus thus would be holding different copy of value returned from reference of A.
    - So in example above we have attained thread-safe class with no explicit locking.
* **Safe Publication:**
  + So far we are not publishing the object. That is we are trying to make the object invisible to other threads and which by change use any reference to the object we want it to be immutable object so that there is no problems with multiple threads using it.
  + static initializers are the easiest and safest way to push an object.
    - public static SomeClass someClass = new SomeClass(“This is Some Class”);
    - JVM executes this statement at class initialization, and because of internal synchronization of JVM, this mechanism is guaranteed to safely publish the object initialized this way.
  + In case of mutable object, we need to take care whenever the object is accessed or modified it should be done synchronously.
    - guarded by a lock.
  + It should always be documented how a published object is going to be accessed.

**Composing Objects:**

Composing classes so that is makes it easier to make them thread-safe.

***Designing a Thread safe class:***

* one general rule is encapsulation helps in thread safety as we can verify and determine the thread safety without going through whole of the program.
* All the primitive and reference type that are encapsulated in a class determines the states of the class.
* Synchronization policy for a class is how the access is there for the states of the class.
* **Synchronization Requirements:**
  + state space: domain and range of the state variable
  + postCondition: state transitions those are valid or invalid.
  + multivariate invariants create atomicity requirements.
    - as the values for multiple variables together makes sense and otherwise it would create race conditions for different variables.
* **State-dependent Operations:**
  + state-dependent operations which required the preconditions to be fulfilled
    - a case like removing element from a queue. The requirement is the queue must contain elements.
  + In threaded application the element might be pushed into the queue by another thread. So the current thread might have to wait for the element to be pushed into the queue.
  + *wait* and *notify is one of the*build-in mechanism to support waiting for a condition to become true.
* **State ownership:**
  + state ownership and encapsulation goes hand in hand. Though some mutable object might skip this ownership and the object might have shared ownership for that state. A case of split ownership.
  + Collections often have split ownership.

**Instance Confinement:**

* instance confinement which is guaranteed by encapsulation.
* confinement + locking can help in using non-thread safe objects in a thread safe manner.
* so basically it means that if we are having a reference to the class object of one type as private field.
  + we can create synchronized(atomic) operations (methods) around it so that we expose only the information that is required and that would make the encapsulating class as thread-safe.
* There are some Collections(Collections.synchronizedList) that already provides the functionality.
  + these are like wrapper to usually collection that would make the class encapsulating variable of this type thread-safe.
  + Though all the operations must be performed using these wrapper classes only, so that thread-safe behavior is guaranteed.
* Again confined objects/instances can escaped if the are not properly published and this may cause some instance to escape thus compromising thread-safe behavior of the encapsulating class.
* ***Java Monitor Patten***
  + This pattern is simple. We guard the mutable variables in the class using a lock. Any lock object can be used.
  + The object used for locking can be a private object or a public object. Though private is preferred as no one else can modify it. Like the client program.
  + **NOTE:**  A good pattern to make the mutable objects as thread safe is to always return the copy of mutable items in the get method and to save the copy of received items in the set method.
    - using this there are no change for the mutable object to get escape.
    - This kind of pattern is followed when we are doing get/set with the Date type of variables in a class.

**Task Execution:**

* A simpel scenario where we are giving a high computational task to a new thread. Assuming the the high computational task is independent.
  + All the tasks are going to execute in parallel thus increasing the performance.
  + Though there are some drawbacks
    - Thread lifecycle overhead
      * Thread creating and destruction has computational overhead and might not be good for very light requests.
    - Resource consumption
      * Threads consume system resources.
      * When there are more threads than the available processors, the threads sit idle thus consuming more memory. Thus they would pressurize the garbage collector
      * If all CPU are already busy with the current threads, creating new threads might degrade the performance.
    - Stability
      * Creating more threads than permitted by OS would raise OutOfMemory Exception and it’s difficult to recover from the error.
  + So conclusion is there should be way to configure how many threads should be created.
* ***Executor Framework:***
  + provides a flexible implementation of threadPool
  + It follows the producer-consumer pattern where the submitter task is the producer which create tasks and the thread that executes the task is the consumer.
  + We can implement the Executor to change the configuration of how the threads are assigned.
  + We can create a new thread for each new request or a single thread for all.
  + **Execution Policies:**
    - what, where, when and how the task would be executed.
  + **Thread Pool:**
    - homogeneous pool of worker threads
    - life cycle of worker threads
      * request **work queue** for the next task
      * execute it.
      * go back waiting for another task.
    - Advantages over creating new threads for each work
      * limits the number of threads created thus lowers creation and teardown overhead of threads.
    - Types of TheadPools:
      * FixedSizedThreadPool - guessable
      * newCachedThreadPool
        + kills the threads if they are more than the current requirement
        + spawn new threads when the requirement increases.
        + It doesn’t case for the limit and thus gives outOfMemory Exception if too many threads are spawned.
      * SingleThreadExecutor
      * ScheduledThreadPool
        + fixed sized thread pool that supports delayed and periodic task execution (like Timer).
  + In pool based policy the web server doesn’t fail under heavy load.
  + **Executor Life Cycle:**
    - how to shut down the executor
    - JVM can’t exit unless all the crated threads are terminated
    - ExecutorService is the class that extends the Executor and adds number of methods for lifecycle management.
      * Stages
        + running
        + shutting down
        + terminated
  + **Delayed and periodic tasks**
    - Timer vs ScheduleThreadPoolExecutor
      * Timer uses a single thread for executing timer tasks.
      * ScheduledThreadPoolExecutor instead uses multiple threads for differed and periodic tasks.
    - Timer behaves very badly when an unchecked exception is thrower from the the TimerTask.
      * Timer doesn’t catch the uncheck exception and thus terminates.
      * Timer doesn’t recurrent the thread in this case and all the scheduled by not started tasks never run.
      * Also no new tasks can be scheduled now
      * **This is known as thread leakage**.
  + **Find parallelism:**
    - Result bearing tasks: Callable and Future
      * Executor uses Runnable as it’s basic task representation
        + Though Runnable is a fairly limited abstraction as it can’t return any values or throw checked exception.
        + Though it can make change in a shared data structure that can be read by other thread.
      * In case of differed calls, Callable is a better abstraction.
        + it expects that the main entry point ***call*** will return a value and might throw an exception(for non returning task it uses Callable<void>)
    - The task executed by the Executor has 4 phases:
      * created
      * submitted
      * started
      * completed/
    - ***In Executor framework, tasks that are started can sometime be cancelled if they are responsive to interruption***.
    - Future represents the lifecycle of a task.
      * provides methods to test whether the task has completed or been cancelled, retrieve it’s result and cancel the task.
      * returns immediately or throws an exception if the task has already completed.
        + if not then it’s blocked unless the task is completed.
        + if exception is throws it’s wrapped in Execution-Exception.

if it’s cancelled - CancellationException

in ExecutionException that underlying cause can be retrieved using getCause().

* + - submitting a runnable or callable to executor and setting the result value for a future from the thread where the task was completed constitutes a safe publication.
  + **CompletionService:**
    - a situation arises where we have a batch task and we want to retrieve the results as they become available
      * As Future.get() is a blocking call. We can do get using a timeout of zero so that we can poll again and again for the completion of the tasks.
      * Another way to tackle this problem is using a CompletionService.
    - it combines the functionality of a blockingQueue and an Executor Service.
    - we can submit a callable task, use the queue like method of poll and take to retrieve the completed results which are packed in Future object as they become available.
    - ExectuorCompletionService implements CompletionService - delegates the computation to the Executor.
    - Implementation:
      * implementation is quite simple
      * the constructor creates a blocking queue, to hold the completed results.
      * FutureTask has **done** method which is called when the task is completed.
      * When the task is submitted it’s wrapped in QueueingFuture, which is a subclass of FutureTask. It overrides the done method to place it inside a BlockingQueue.
      * The **take** and **poll** methods delegates to the BlockingQueue. Thus blocking if the results are not available.
    - How this can be helpful?
      * Consider an application of rendering an html page with test and images. Amazon retain home page.
      * There are different images and text to be downloaded. Let’s assume are calling multiple services for different txt and images that are required to be rendered on the page.
      * We can call all those services in parallel using ExecutorCompletionService and wait for each of the parallel task to be completed.
      * As they become available on the BlockingQueue for rendering we would render them on the html page one by one.
      * Thus making the rendering of a webpage in amazon as fast as possible.
  + Future.get() takes the timeOut and thus raises TimeoutException if the result is not ready within a given time period.
    - This raises the exception but the task might still be running on the thread.
  + So another problem is to stop the timed task when they are running out of time.
    - We can create the task such that it’s managing it’s own time budget and aborts if it runs out of time.
    - or by cancelling the task if the time expires.
    - Sometime the task might not be written to be cancellable, it can be terminated early so that it doesn’t consume extra resources
      * this is to be covered in next chapter.
  + InvokeAll can take a list of Tasks and return a list of future in the same order. Thus invoking all those tasks at once.

**Cancellation and ShutDown:**

* Till now the focus was on letting the task complete naturally, that is reaching their end point.
* But in many situations we would like the task to stop.
* For example I am parallelizing 10 api calls together
  + But I am processing the results from the calls one by one.
  + As a point I get some exception where I have to stop the future processing.
  + As a result I would want to stop all the processing thread of the remaining 9 api calls if they are still processing.
  + Because they might be occupying extra processing and memory for the task executing which is no longer needed.
* Java doesn’t provide any mechanism for safely forcing a thread to stop what it’s doing.
* Instead it provides interruption.
  + that would tell the thread to stop what it’s doing.
* Why this approach is required?
  + because stopping immediately might leave some shared data structure in an inconsistent state.
* so it’s better to clean the current work and then terminate.
* ***Task Cancellation:***
  + cancellation request flag
    - that task periodically check this flag
    - if it’s set then the task terminates early.
    - It would be like when u are processing a heavy task in loop. Check every time for the flag to see if it’s true or not.
    - It’s good to keep an infinite running task use this flag. So that when we reach a condition we can use this flag to stop the thread from executing.
      * As otherwise an infinite loop is going to consume CPU cycles.
  + A task that wants to be cancelled should have a cancellation policy
  + A simple case where the cancellation request flag fails is the case when the thread that is required to be cancelled is waiting for the blocking queue.
  + Here comes the interrupt into picture.
    - A thread can listen and act on the interruption at it’s own will.
    - That is the other thread can raise an interruption but the running thread might stop running or can continue it’s processing which depends on the implementation of the task being run.
  + ***interrupted status***
    - Each thread has a boolean interrupted status.
    - interrupt()
      * will interrupt the thread, thus setting the interrupted status flag
      * isInterrupted()
        + reads the interrupted flag and checks whether thread is interrupted or not.
    - We can do a check on isInterrupted() and catch the InterruptedException.
  + For a Thread there should be an interruption policy.
  + ***interruption Policies:***
    - important to distinguish how task and thread reacts to interruption.
    - A single interrupt might have more than one desired recipient
      * the worker thread.
      * the task.
    - Tasks don’t execute on their own thread. They borrow the thread from the worker thread pool.
    - it means that the code of task doesn’t own the thread.
      * It should carefully preserve the interruption status so that the owning code eventually acts on it, even if the guest code acts on the interruption.
    - So when a task sees an InterruptionException, it can postpone it until it’s handling the exception. It can finish the task it was performing and then throw the InterruptedException or otherwise indicate interrupt.
    - A task should not assume anything about the executing thread. The task can propagate the interruptedException to it’s working thread or using currentThread().interrupt() to handle the finish, finish the incomplete task at some logical point and then raise it again.
      * We can retrain to call interrupt() again if the case is when the task code is actually handling the interruption policy for a thread.
      * Most code doesn’t know what thread it will run in so its best to preserve the interrupt status and propagate it or raise it again using interrupt().
  + ***Cancellation using Future***
    - Future has cancel(mayInterruptIfRunning boolean)
      * returns a value indicating whether the attempt was successful
      * tells only if it was able to deliver the interruption
        + not whether the task detected and acted on it.
      * when mayInterruptedIfRunning is true and the  task is currently running in some threads, it means that the thread is interrupted
      * if it’s false it mean don’t run the task if it hasn’t started yet.
        + should be used for the tasks that are not designed to handle interruptions
    - generally we should not interrupt a thread unless we know it’s interruption policy, which in this case? When we are supplying an argument of true to the cancel()
      * In task execution, thread created by the standard executor implements an interruption policy that lets the tasks be cancelled using interruption.
      * Though we should not interrupt a pool thread directly when attempting to cancel a task as we won’t know what task is running when the interrupt request is delivered.
        + **do this only through the task’s future.**
      * cancelling a completed task has no effect.
  + ***Non-Interruptible blocking***
    - Many blocking libraries respond to blocking by **throwing** **InterruptedException**.
    - Though there are some which don’t respond like sync socket I/O or waiting for intrinsic lock
      * In such cases the  interruption has no effect other than setting the thread’s interrupted status
    - In case of Sync I/O socket waiting, one way is that to override the interrupt method and close the socket on interrupt.
      * once the socket is closed the Thread responds to the interrupt.
  + ***Refining nonstandard cancellation using newTaskFor***
    - newTaskFor creates a future representing the task
      * this future is returned when a callable task is submitted to ExecutorService
    - newTaskFor returns RunnableFuture(implemented by FutureTask).
    - we can customize task Future and override Future.cancel.
      * here we can cancel activities which are not responsive to interruption.
        + like using libraries for closing socket connection for the blocking call, which throw InterruptedException.
* ***Stopping a Thread Based Service:***
  + Encapsulation tells that we should not interrupt a thread, prioritize it unless we own it.
  + Thread API in java has no ownership concept.
  + Though the class that creates the threads can be called owner of those threads.
  + ThreadPool owns its worker threads and if those threads are need to be interrupted it should be through the threadPool.
  + As the ownership of Threads is not transitive it can be said that the application doesn’t own the threads directly rather the service which is owned by the application owns the threads.
    - An application can provides the lifecycle methods that can shutdown the threads through the threadService when it’s shutting itself down.
  + **LifeCycle methods should be provided when the service owning the threads has longer lifetime than the method that created it.**
  + **ExecutorService** provides shutdown and shutdownNow.
    - shutdown waits for the tasks to be completed. This is safer because it waits for the execution completion and then termites the thread.
    - shutdownNow exits the by giving a list of tasks that are not started and tried to cancel the executing tasks.
      * tasks may be interrupted in the middle of execution if they are handling the interruption.
  + When an executor service is being closed there is no direct way of finding out whether all tasks completed successfully or not.
  + We can create a trackingExecutor by extending the abstract executorSerivce, where we override execute() to finally track if the task is shutDown() and is interrupted.
    - Still this could be failure as the thread pool could be shut down in between the last instructions of the tasks executes and the pool can’t record the task completion, thus giving false positive.
    - Though this problem would not big if the tasks are idempotent.
* ***Handling abnormal Thread Termination***
  + As if thread fails there might be conditions that we miss the failure scenarios. Like what is printed in the log.
  + There are some ways of detecting thread’s failure
  + The leading cause if RuntimeException(As thread fails and its non recoverable)
  + Loosing a thread from the thread pool can have bad consequences on the performance of the application.
  + A way that the ThreadPool implements the solution of the problem is using a worker thread within the threadPool.
    - if a task throws an unchecked exception it allows the thread to die but before that in finally block it specified the framework that the thread has died using the worker thread.
    - using the new thread can be created if any thread dies.
  + **Using UncaughtExceptionHandler**
    - Another was is using the UncaughtExceptionHandler provided by the Thread API.
    - when a thread exits due to an uncaughtException, the JVM repots this event to an application-provided UncaughtExceptionHandler.
    - By default it throws the stack trace to System.err but we can override it to print the log.
    - **NOTE:**
      * The tasks that are submitted through the execute are the only one which makes to the uncaughtException
      * The tasks submitted with submit, any thrown exception, checked or not, is considered to be a part of Future.get, wrapped din ExecutionException.
* ***JVM Shutdown:***
  + JVM can shutdown abruptly or orderly.
    - orderly is executed when the last normal thread terminates(nondaemon thread)
  + Orderly shut
    - JVM starts all register shutdown hooks(unstarted threads that are registered with Runtime.addShutDownHook).
      * this is unorderly
    - if any application threads are running in the shutdown time, they continue to run concurrently with the shutdown process.
    - When shutdown hooks are completed the JVM make choose to run finalizers(if runFinalizersOnExit is true) and then halts.
    - If any application threads are still running at shutdown time, they are abruptly terminated and the JVM halts.
    - If shutdown hooks and finalizers don’t complete the JVM orderly process hangs and it must be shutdown abruptly.
  + The shutdown hooks doesn’t run in abrupt shutdown of JVM.
  + ShutDownHooks can be used for cleaning the service before terminating the JVM.
    - They should be thread-safe and we should be careful to prevent any deadlock.
  + **Daemon Threads**
    - when we want to create some helper threads that perform some helper function but don’e want the existence of this thread to prevent JVM from shutting the, Daemon threads come into picture.
    - an example is garbage collector. So thread which are not critical and those termination in middle doesn’t matter can be used as a Daemon Thread.
  + Avoid finalizers!!