JAVA MULTITHREADING

By Vipul Kumar Singh

FEBRUARY 25, 2022

VKS TECH

INDEX

MULTITHREADING IN JAVA

1. Thread

- A thread is a **lightweight sub-process, the smallest unit of processing**.
- Threads are independent. If there occurs exception in one thread, it doesn't affect other threads.
- Threads uses a shared memory area.
- A thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

1.1 Context-Switch

- A context switch is the **process of storing the state of a process or thread**, so that it can be restored and resume execution at a later point.
- This allows multiple processes to share a single central processing unit (CPU), and is an essential feature of a multitasking operating system.

2. Multi-threading

- Multithreading 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.
- We use multitasking to utilize the CPU.

2.1. Multithreading in Java

- Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.
- Each part of such program is called a thread.

3. Ways to create thread in Java

- Threads can be created by using two mechanisms:
  - Extending the Thread class.
  - Implementing the Runnable Interface

3.1. Thread creation by extending the Thread class

- We create a class that extends the java.lang.Thread class.
- This class overrides the run() method available in the Thread class.
- We create an object of our new class and call start() method to start the execution of a thread.
- start() invokes the run() method on the Thread object.

```
class MultithreadingDemo extends Thread {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
                + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
public class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            MultithreadingDemo object
                = new MultithreadingDemo();
            object.start();
        }
    }
}
```

3.1.1. Java 8 Thread lambda syntax

```
Thread t = new Thread(() -> {
    // your code here ...
});
t.start();
```

Or

```
new Thread(() -> // your code here).start();
```

3.2. Thread creation by extending the Thread class

- We create a new class which implements java.lang.Runnable interface and override run() method.
- Then we instantiate a Thread object and call start() method on this object.

```
class MultithreadingDemo implements Runnable {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
                + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            Thread object
                = new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}
```

3.2.1. Java 8 Runnable lambda syntax

```
Runnable runnable = () -> {
    // your code here ...
};
Thread t = new Thread(runnable);
t.start();
```

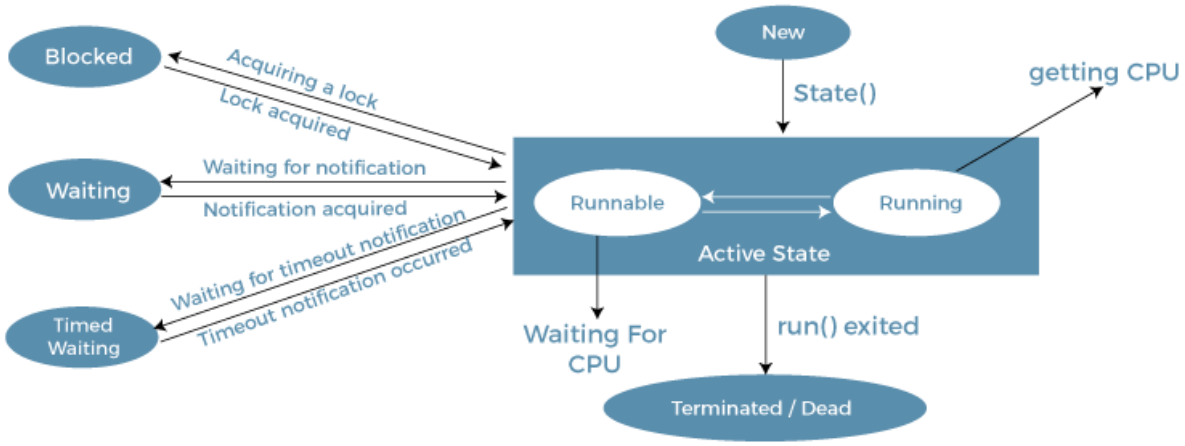
3.3 Runnable vs Thread

- If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
- We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.
- Using runnable will give you an object that can be shared amongst multiple threads.

4. Lifecycle and states of a thread in Java

- A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New
2. Runnable
3. Blocked
4. Waiting
5. Timed Waiting
6. Terminated



Life Cycle of a Thread

4.1. Life Cycle of a thread

- New Thread: When a new thread is created, it is in the new state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.
- Runnable State: A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.
  A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.
- Blocked/Waiting state: When a thread is temporarily inactive, then it's in one of the following states:
  - Blocked
  - Waiting

  A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states does not consume any CPU cycle.

A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the schedule picks one of the thread which is blocked for that section and moves it to the runnable state.

Whereas, a thread is in the waiting state when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.

- Timed Waiting: A thread lies in timed waiting state when it calls a method with a time out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep() or a conditional wait(), it is moved to a timed waiting state.
- Terminated State: A thread terminates because of either of the following reasons:
    - Because it exists normally. This happens when the code of thread has entirely executed by the program.
    - Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

  A thread that lies in a terminated state does no longer consumes any cycles of CPU.

4.2. Implementation of Thread States

In Java, one can get the current state of a thread using the **Thread.getState()** method. The **java.lang.Thread.State** class of Java provides the constants ENUM to represent the state of a thread. These constants are:

- NEW: Thread state for a thread which has not yet started.
- RUNNABLE: It represents the runnable state. It means a thread is waiting in the queue to run.
- BLOCKED: It represents the blocked state. In this state, the thread is waiting to acquire a lock.
- WAITING: It represents the waiting state. A thread will go to this state when it invokes the Object.wait() method, or Thread.join() method with no timeout. A thread in the waiting state is waiting for another thread to complete its task.
- TIMED_WAITING: It represents the timed waiting state. The main difference between waiting and timed waiting is the time constraint. Waiting has no time constraint, whereas timed waiting has the time constraint. A thread invoking the following method reaches the timed waiting state.
    - sleep
    - join with timeout
    - wait with timeout
    - parkUntil
    - parkNanos
- TERMINATED: It represents the final state of a thread that is terminated or dead.

5. Constructors of Thread Class

- Thread()
- Thread(Runnable target)
- Thread(Runnable target, String name)
- Thread(String name)
- Thread(ThreadGroup group, Runnable target)
- Thread(ThreadGroup group, Runnable target, String name)
- Thread(ThreadGroup group, Runnable target, String name, long stackSize)
- Thread(ThreadGroup group, String name)

6. ThreadGroup

- Java provides a convenient way to group multiple threads in a single object. In such a way, we can suspend, resume or interrupt a group of threads by a single method call.
- Java thread group is implemented by java.lang.ThreadGroup class.
- A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.
- A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

```
ThreadGroup tg1 = new ThreadGroup("Group A");
Thread t1 = new Thread(tg1,new MyRunnable(),"one");
Thread t2 = new Thread(tg1,new MyRunnable(),"two");
Thread t3 = new Thread(tg1,new MyRunnable(),"three");
```

Now we can interrupt all threads by a single line of code only.

```
Thread.currentThread().getThreadGroup().interrupt();
```

7. Methods of Thread Class

- **public static int activeCount() :** Returns an estimate of the number of active threads in the current thread's thread group and its subgroups
- **checkAccess():** Determines if the currently running thread has permission to modify this thread
- **clone():** Throws CloneNotSupportedException as a Thread cannot be meaningfully cloned
- **currentThread():** Returns a reference to the currently executing thread object
- **dumpStack():** Prints a stack trace of the current thread to the standard error stream
- **enumerate(Thread[] tarray):** Copies into the specified array every active thread in the current thread's thread group and its subgroups
- **getAllStackTraces():** Returns a map of stack traces for all live threads
- **getContextClassLoader():** Returns the context ClassLoader for this Thread
- **getDefaultUncaughtExceptionHandler():** Returns the default handler invoked when a thread abruptly terminates due to an uncaught exception
- **getId():** Returns the identifier of this Thread
- **getName():** Returns this thread's name
- **getPriority():** Returns this thread's priority
- **getStackTrace():** Returns an array of stack trace elements representing the stack dump of this thread
- **getState():** Returns the state of this thread

- **getThreadGroup():** Returns the thread group to which this thread belongs
- **getUncaughtExceptionHandler():** Returns the handler invoked when this thread abruptly terminates due to an uncaught exception
- **holdsLock(Object obj):** Returns true if and only if the current thread holds the monitor lock on the specified object
- **interrupt():** Interrupts this thread
- **interrupted():** Tests whether the current thread has been interrupted
- **isAlive():** Tests if this thread is alive
- **isDaemon():** Tests if this thread is a daemon thread
- **isInterrupted():** Tests whether this thread has been interrupted
- **join():** Waits for this thread to die
- **join(long millis):** Waits at most millis milliseconds for this thread to die
- **run():** If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns
- **setContextClassLoader(ClassLoader cl):** Sets the context ClassLoader for this Thread
- **setDaemon(boolean on):** Marks this thread as either a daemon thread or a user thread
- **setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh):** Set the default handler invoked when a thread abruptly terminates due to an uncaught exception, and no other handler has been defined for that thread
- **setName(String name):** Changes the name of this thread to be equal to the argument name.
- **setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh):** Set the handler invoked when this thread abruptly terminates due to an uncaught exception
- **setPriority(int newPriority):** Changes the priority of this thread
- **sleep(long millis):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers
- **start():** Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread
- **toString():** Returns a string representation of this thread, including the thread's name, priority, and thread group
- **yield():** A hint to the scheduler that the current thread is willing to yield its current use of a processor

8. Joining threads in Java

- Sometimes one thread needs to know when other thread is terminating.
- In java, isAlive() and join() are two different methods that are used to check whether a thread has finished its execution or not.
- The isAlive() method returns true if the thread upon which it is called is still running otherwise it returns false.

```
final boolean isAlive()
```

- But, **join()** method is used more commonly than **isAlive()**. This method waits until the thread on which it is called terminates.

```
final void join() throws InterruptedException
```

- Using **join()** method, we tell our thread to wait until the specified thread completes its execution.
- There are overloaded versions of **join()** method, which allows us to specify time for which you want to wait for the specified thread to terminate.

```
final void join(long milliseconds) throws InterruptedException
```

9. Inter-thread communication

- Inter-thread communication in Java is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

9.1. What is Polling, and what are the problems with it?

- The process of testing a condition repeatedly till it becomes true is known as polling.
- Polling is usually implemented with the help of loops to check whether a particular condition is true or not. If it is true, a certain action is taken.
- This wastes many CPU cycles and makes the implementation inefficient.
- To avoid polling, Java uses **wait** and **notify** methods.

9.2. wait()

- It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().
- The wait() method releases the lock prior to waiting and reacquires the lock prior to returning from the wait() method.
- The wait() method is actually tightly integrated with the synchronization lock, using a feature not available directly from the synchronization mechanism.
- In other words, it is not possible for us to implement the wait() method purely in Java. It is a native method.
- General syntax for calling wait() method is like this:

```
synchronized( lockObject )
{
    while( ! condition )
    {
        lockObject.wait();
    }

    //take the action here;
}
```

- wait() has 3 variations:
    - public final void wait(): will cause thread to wait till notify is called
    - public final void wait(long timeout): will cause thread to wait either till notify is called or till timeout (One which occurs earlier)
    - public final void wait(long timeout, int nanoseconds): takes a timeout argument as well as a nanosecond argument for extra precision
- It is mandatory to enclose wait() in a try-catch block because if a thread present in the waiting state gets interrupted, then it will throw InterruptedException.
- The other two variations of wait housing parameters will throw IllegalArgumentException if the value of timeout is negative or the value of nanoseconds is not in the range 0 to 9,99,999
- `wait()` method must occur inside a synchronized block of code, if not, an `IllegalMonitor` exception is raised.
- `wait()` method should occur inside a loop, because we will encounter the **Spurious Wakeups** problem. It means that a thread is woken up even though no signal has been received. **Spurious wakeups** are a reality and are one of the reasons why the pattern for waiting on a condition variable happens in a while loop.

9.3. notify()

- It wakes up one single thread called wait() on the same object.
- It should be noted that calling notify() does not give up a lock on a resource.
- It tells a waiting thread that that thread can wake up. However, the lock is not actually given up until the notifier's synchronized block has completed.
- General syntax for calling notify() method is like this:

```
synchronized(lockObject)
{
    //establish_the_condition;

    lockObject.notify();

    //any additional code if needed
}
```

- notify() has 1 signature:
    - public final void notify()
- When the notify() is called on a thread holding the monitor lock, it symbolizes that the thread is soon going to surrender the lock. One of the waiting threads is randomly selected and notified about the same. The notified thread then exits the waiting state and enters the blocked state where it waits till the previous thread has given up the lock and this thread has acquired it. Once it acquires the lock, it enters the runnable state where it waits for CPU time and then it starts running.

9.4. notifyAll()

- It wakes up all the threads that called wait() on the same object. The highest priority thread will run first in most of the situation, though not guaranteed. Other things are same as notify() method above.
- General syntax for calling notify() method is like this:

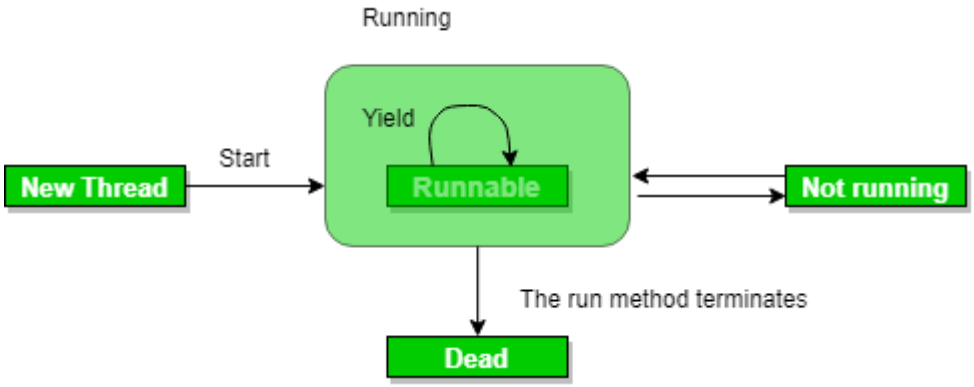
```
synchronized(lockObject)
{
    establish_the_condition;

    lockObject.notifyAll();
}
```

9.5. wait(long timeout) vs sleep(long timeout)

- wait(timeout): when the timeout expires the thread wakes up and tries to re-acquire the synchronization lock (i.e. if another thread has not notified it within the timeout period).
- sleep(timeout): sleep can be used without any sycnhronization code - it just puts the thread to sleep for the specified timeout. Note that wait must be used within synchronized code.
- So, wait is used when you expect a thread to be notified by another thread (but may not, hence timeout). And, wait must be called within synchronized code.

9.6. yield() method

- **yield()** basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should run. Otherwise, the current thread will continue to run.



- If any thread executes yield method, thread scheduler checks if there is any thread with same or high priority than this thread. If processor finds any thread with higher or same priority then it will move the current thread to Ready/Runnable state and give processor to other thread and if not – current thread will keep executing.
- Once a thread pauses its execution, we can't specify when it will get chance again it depends on thread scheduler.
- Underlying platform must provide support for preemptive scheduling if we are using yield method.

10. Synchronization In Java

- Synchronization in java is the capability to control the access of multiple threads to any shared resource.
- In the Multithreading concept, multiple threads try to access the shared resources at a time to produce inconsistent results.
- The synchronization is necessary for reliable communication between threads.

10.1. Synchronized Method

- If we use the Synchronized keywords in any method then that method is Synchronized Method.
- It is used to lock an object for any shared resources.
- The object gets the lock when the synchronized method is called.
- The lock won't be released until the thread completes its function.
- Syntax:

```
acessModifiers synchronized returnType methodName
(MethodParameters) {
        // Code of the Method.
}
```

- It is not possible for two invocations for synchronized methods to interleave. If one thread is executing the synchronized method, all others thread that invoke synchronized method on the same Object will have to wait until first thread is done with the Object.

10.2. Synchronized Block

- If we only need to execute some subsequent lines of code not all lines (instructions) of code within a method, then we should synchronize only block of the code within which required instructions are exists.
- Syntax:

```
acessModifiers returnType methodName (MethodParameters) {
     .
     .
     synchronized (object){
         // Code of the Block
     }
     .
     .
}
```

10.3. Static Synchronization

- In java, every object has a single lock (monitor) associated with it. The thread which is entering into synchronized method or synchronized block will get that lock, all other threads which are remaining to use the shared resources have to wait for the completion of the first thread and release of the lock.
- Suppose in the case of where we have more than one object, in this case, two separate threads will acquire the locks and enter into a synchronized block or synchronized method with a separate lock for each object at the same time. To avoid this, we will use static synchronization.
- In this, we will place synchronized keywords before the static method.
- In static synchronization, lock access is on the class not on object and Method.
- Syntax:

```
synchronized static returnType methodName (Parameters) {
    //code
}

Or

synchronized static returnType methodName (ClassName.class) {
    //code
}
```

10.4. Important Points

- When a thread enters into synchronized method or block, it acquires lock and once it completes its task and exits from the synchronized method, it releases the lock.
- When thread enters into synchronized instance method or block, it acquires Object level lock and when it enters into synchronized static method or block it acquires class level lock.
- Java synchronization will throw null pointer exception if Object used in synchronized block is null. For example, If in synchronized(instance) , instance is null then it will throw null pointer exception.
- In Java, wait(), notify() and notifyAll() are the important methods that are used in synchronization.

- You can not apply java synchronized keyword with the variables.
- Don’t synchronize on the non-final field on synchronized block because the reference to the non-final field may change anytime and then different threads might synchronize on different objects i.e. no synchronization at all.

10.5. Advantages

- **Multithreading:** Since java is multithreaded language, synchronization is a good way to achieve mutual exclusion on shared resource(s).
- **Instance and Static Methods:** Both synchronized instance methods and synchronized static methods can be executed concurrently because they are used to lock different Objects.

10.6. Disadvantages

- **Concurrency Limitations:** Java synchronization does not allow concurrent reads.
- **Decreases Efficiency:** Java synchronized method run very slowly and can degrade the performance, so you should synchronize the method when it is absolutely necessary otherwise not and to synchronize block only for critical section of the code.

11. Java volatile keyword

- Using volatile is yet another way (like synchronized, atomic wrapper) of making class thread safe.
- If two threads run on different processors each thread may have its own local copy of **shared variable**. If one thread modifies its value the change might not reflect in the original one in the main memory instantly. Now the other thread is not aware of the modified value which leads to data inconsistency.
- The solution to above problem is the volatile keyword.
- The Java `volatile` keyword guarantees visibility of changes to variables across threads.
- Example:

```
class SharedObj
{
   // volatile keyword here makes sure that
   // the changes made in one thread are
   // immediately reflect in other thread
   static volatile int sharedVar = 6;
}
```

11.1. volatile vs synchronized

- Two important features of locks and synchronization are:
    - Mutual Exclusion: It means that only one thread or process can execute a block of code (critical section) at a time.
    - Visibility: It means that changes made by one thread to shared data are visible to other threads.
- Java's synchronized keyword guarantees both mutual exclusion and visibility.
- If we make the blocks of threads that modifies the value of shared variable synchronized only one thread can enter the block and changes made by it will be reflected in the main memory. All other thread trying to enter the block at the same time will be blocked and put to sleep.
- In some cases, we may only desire visibility and not atomicity. Use of synchronized in such situation is an overkill and may cause scalability problems. Here volatile comes to the rescue.
- Volatile variables have the visibility features of synchronized but not the atomicity features.
- The values of volatile variable will never be cached and all writes and reads will be done to and from the main memory.
- However, use of volatile is limited to very restricted set of cases as most of the times atomicity is desired.

11.2. volatile vs static

- A static variable in Java is stored once per class (not once per object, such as non-static variables are). This means all your objects (and static methods) share the same variable.
- Declaring a variable as volatile (be it static or not) states that the variable will be accessed frequently by multiple threads. In Java, this boils down to instructing threads that they cannot cache the variable's value, but will have to write back immediately after mutating so that other threads see the change. (Threads in Java are free to cache variables by default).
- Declaring a static variable in Java, means that there will be only one copy, no matter how many objects of the class are created. The variable will be accessible even with no Objects created at all. However, threads may have locally cached values of it.
- When a variable is volatile and not static, there will be one variable for each Object. So, on the surface it seems there is no difference from a normal variable but totally different from static. However, even with Object fields, a thread may cache a variable value locally.
- This means that if two threads update a variable of the same Object concurrently, and the variable is not declared volatile, there could be a case in which one of the threads has in cache an old value.
- Even if you access a static value through multiple threads, each thread can have its local cached copy. To avoid this, you can declare the variable as static volatile and this will force the thread to read each time the global value.
- However, volatile is not a substitute for proper synchronisation.

	Shared between objects	**Locally Cached**
static	Yes	Yes
volatile	No	No
static volatile	Yes	No

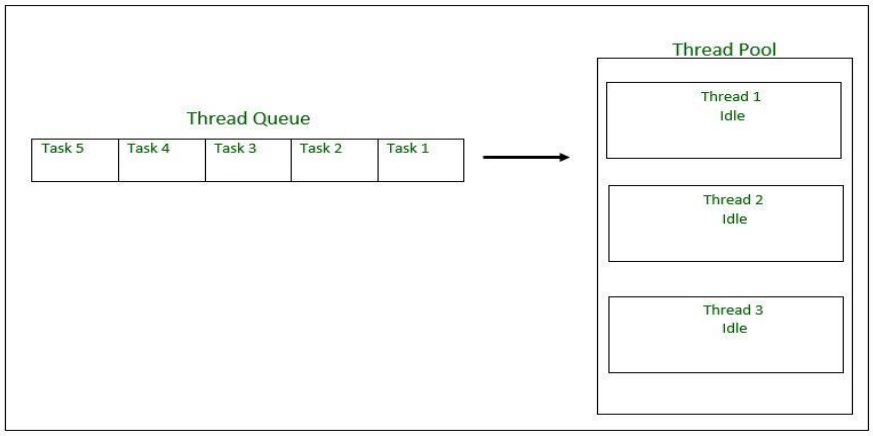
12. Atomic Variables

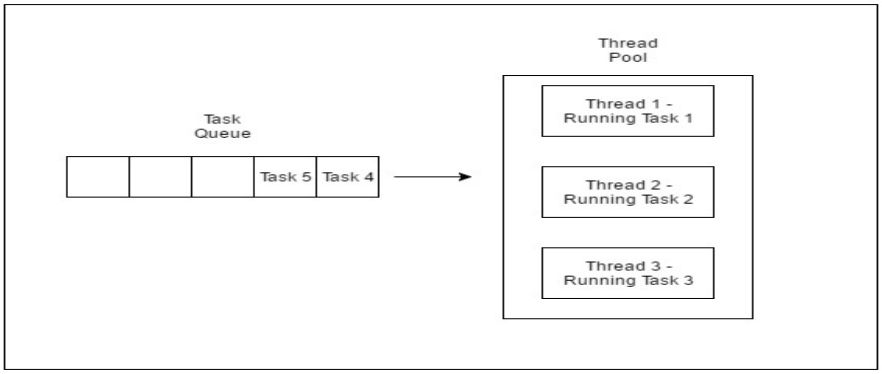
- In multithreading, the shared entity mostly leads to a problem when concurrency is incorporated.
- A shared entity such as, mutable object or variable, might be changed, which may result in the inconsistency of the program or database.
- So, it becomes crucial to deal with the shared entity while accessed concurrently. An atomic variable can be one of the alternatives in such a scenario.
- The atomic classes are **mutable**, but have strong memory consistency guarantees with regard to modifications. So, they serve a different purpose from the immutable wrapper classes.
- The real advantage of the Atomic classes is that they expose an atomic **compareAndSet** method, which can be very useful for implementing lock-free algorithms.
- Java provides atomic classes such as **AtomicInteger**, **AtomicLong**, **AtomicBoolean** and **AtomicReference**. Objects of these classes represent the atomic variable of int, long, boolean, and object reference respectively.
- These classes contain the following methods.
    - **set(int value):** Sets to the given value
    - **get():** Gets the current value
    - **lazySet(int value):** Eventually sets to the given value
    - **compareAndSet(int expect, int update):** Atomically sets the value to the given updated value if the current value == the expected value
    - **addAndGet(int delta):** Atomically adds the given value to the current value
    - **decrementAndGet():** Atomically decrements by one the current value

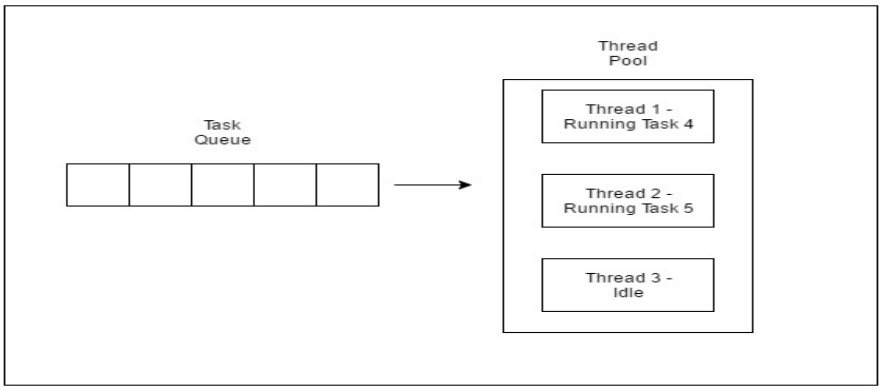
13. ThreadPool

- Active threads consume system resources, so a JVM creating too many threads at the same time can cause the system to run out of memory. This necessitates the need to limit the number of threads being created by an application.
- **A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing.**
- Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.

13.1. Thread pool demonstration

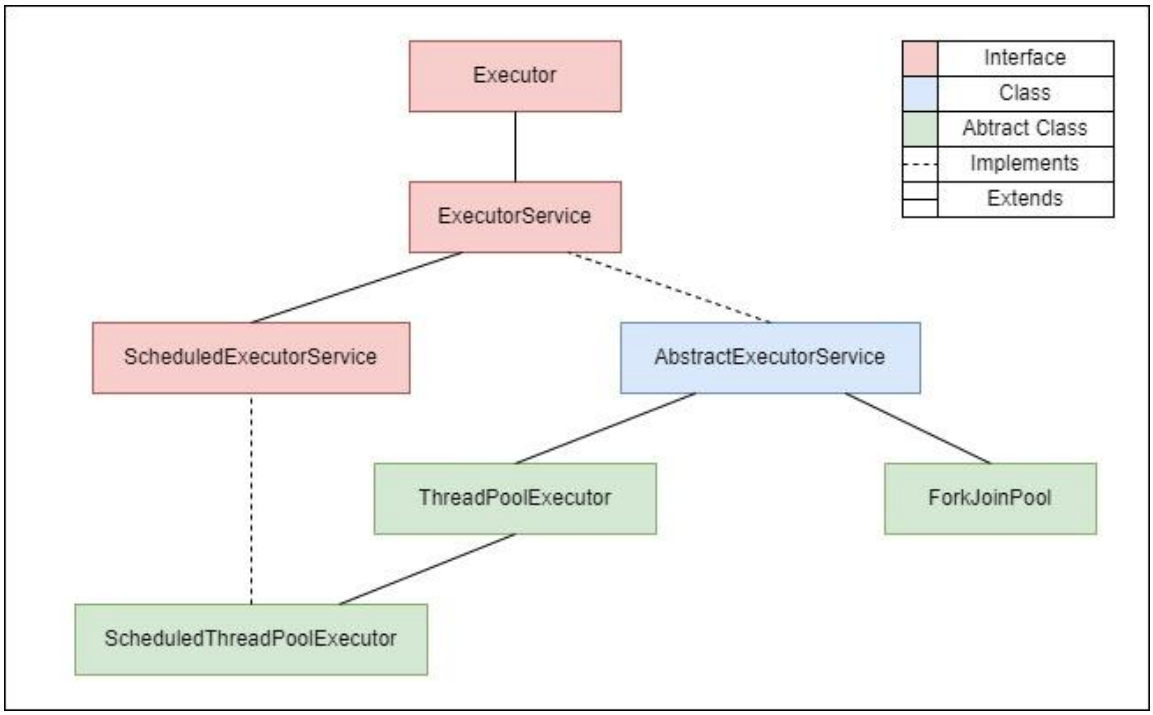






13.2. Thread pool implementations

- Java provides the Executor framework which is centered around the Executor interface, its sub-interface – ExecutorService and the class - ThreadPoolExecutor, which implements both of these interfaces. By using the executor, one only has to implement the Runnable objects and send them to the executor to execute.
- They allow you to take advantage of threading, but focus on the tasks that you want the thread to perform, instead of thread mechanics.
- To use thread pools, we first create a object of ExecutorService and pass a set of tasks to it. ThreadPoolExecutor class allows to set the core and maximum pool size. The runnables that are run by a particular thread are executed sequentially.



13.2.1. Executor

- The Executor interface has a single execute method to submit Runnable instances for execution.
- We can use the Executors API to acquire an Executor instance.

```
Executor executor = Executors.newSingleThreadExecutor();
executor.execute(() -> System.out.println("Hello World"));
```

13.2.2. ExecutorService

- The Executor interface has a single execute method to submit Runnable instances for execution.
- The ExecutorService interface contains a large number of methods to **control the progress of the tasks and manage the termination of the service.**
- Using this interface, we can submit the tasks for execution and also control their execution using the returned Future instance

```
ExecutorService executorService =
Executors.newFixedThreadPool(10);
Future<String> future = executorService.submit(() -> "Hello
World");
// some operations
String result = future.get();
```

```
ExecutorService executorService =
    new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,
    new LinkedBlockingQueue<Runnable>());
```

13.2.3. ThreadPoolExecutor

- The ThreadPoolExecutor is an extensible thread pool implementation with lots of parameters and hooks for fine-tuning.
- The main configuration parameters that we'll discuss here are **corePoolSize**, **maximumPoolSize** and **keepAliveTime.**
    - The **corePoolSize** parameter is the number of core threads that will be instantiated and kept in the pool.
    - When a new task comes in, if all core threads are busy and the internal queue is full, the pool is allowed to grow up to **maximumPoolSize.**
    - The **keepAliveTime** parameter is the interval of time for which the excessive threads (instantiated in excess of the corePoolSize) are allowed to exist in the idle state.
- By default, the ThreadPoolExecutor only considers non-core threads for removal. In order to apply the same removal policy to core threads, we can use the `allowCoreThreadTimeOut(true)` method.

```
ThreadPoolExecutor executor =
   (ThreadPoolExecutor) Executors.newFixedThreadPool(2);
executor.submit(() -> {
    Thread.sleep(1000);
    return null;
})
```

13.2.4. ScheduledThreadPoolExecutor

- The *ScheduledThreadPoolExecutor* extends the *ThreadPoolExecutor* class and also implements the *ScheduledExecutorService* interface with several additional methods:
  - **schedule(**) method allows us to run a task once after a specified delay.
  - **scheduleAtFixedRate()** method allows us to run a task after a specified initial delay and then run it repeatedly with a certain period. The period argument is the time measured between the starting times of the tasks, so the execution rate is fixed.
  - **scheduleWithFixedDelay()** method is similar to scheduleAtFixedRate in that it repeatedly runs the given task, but the specified delay is measured between the end of the previous task and the start of the next. The execution rate may vary depending on the time it takes to run any given task.

```
ScheduledExecutorService executor =
Executors.newScheduledThreadPool(5);
executor.schedule(() -> {
    System.out.println("Hello World");
}, 500, TimeUnit.MILLISECONDS);
```

13.2.5. ForkJoinPool

- *ForkJoinPool* is the central part of the *fork/join* framework introduced in Java 7.
- It solves a common problem of **spawning multiple tasks in recursive algorithms.**
- We'll run out of threads quickly by using a simple *ThreadPoolExecutor*, as every task or subtask requires its own thread to run.
- In a *fork/join* framework, any task can spawn (*fork*) a number of subtasks and wait for their completion using the *join* method.
- The benefit of the *fork/join* framework is that it **does not create a new thread for each task or subtask**, instead implementing the work-stealing algorithm.

13.2.5.1. Work-Stealing Algorithm

- Simply put – free threads try to "steal" work from deques of busy threads.
- By default, a worker thread gets tasks from the head of its own deque. When it is empty, the thread takes a task from the tail of the deque of another busy thread or from the global entry queue, since this is where the biggest pieces of work are likely to be located.
- This approach minimizes the possibility that threads will compete for tasks. It also reduces the number of times the thread will have to go looking for work, as it works on the biggest available chunks of work first.

```
ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();
int sum = forkJoinPool.invoke(new CountingTask(tree));
```

13.3. Executors API

- Factory and utility methods for `Executor`, `ExecutorService`, `ThreadFactory`, `ScheduledExecutorService`, and `Callable` classes.
- The most typical configurations are predefined in the Executors static methods.
- Commonly used methods are :
    - **newFixedThreadPool()**: this method creates a *ThreadPoolExecutor* with equal *corePoolSize* and *maximumPoolSize* parameter values and a zero *keepAliveTime*.
    - **newCachedThreadPool()**: This method does not receive a number of threads at all, but set the corePoolSize to 0 and maximumPoolSize to Integer.MAX_VALUE, and keepAliveTime to 60 seconds.
    - **newSingleThreadExecutor():** Contains a single thread. The single thread executor is ideal for creating an event loop. The *corePoolSize* and *maximumPoolSize* parameters are equal to 1, and the *keepAliveTime* is 0.

13.4. Risks in using Thread Pools

- **Deadlock**: While deadlock can occur in any multi-threaded program, thread pools introduce another case of deadlock, one in which all the executing threads are waiting for the results from the blocked threads waiting in the queue due to the unavailability of threads for execution.
- **Thread Leakage**: Thread Leakage occurs if a thread is removed from the pool to execute a task but not returned to it when the task completed. As an example, if the thread throws an exception and pool class does not catch this exception, then the thread will simply exit, reducing the size of the thread pool by one. If this repeats many times, then the pool would eventually become empty and no threads would be available to execute other requests.
- **Resource Thrashing**: If the thread pool size is very large then time is wasted in context switching between threads. Having more threads than the optimal number may cause starvation problem leading to resource thrashing as explained.

13.5. Tuning Thread Pool

- The optimum size of the thread pool depends on the number of processors available and the nature of the tasks. On a N processor system for a queue of only computation type processes, a maximum thread pool size of N or N+1 will achieve the maximum efficiency.But tasks may wait for I/O and in such a case we take into account the ratio of waiting time(W) and service time(S) for a request; resulting in a maximum pool size of N*(1+ W/S) for maximum efficiency.

14. Callable and Future

14.1. Callable

- One feature lacking in Runnable is that we cannot make a thread return result when it terminates, i.e. when run() completes. For supporting this feature, the Callable interface is present in Java.
- For implementing Runnable, the run() method needs to be implemented which does not return anything, while for a Callable, the call() method needs to be implemented which returns a result on completion.
- Note that a thread can't be created with a Callable, it can only be created with a Runnable.
- Another difference is that the call() method can throw an exception whereas run() cannot.

```
class CallableExample implements Callable{
  public Object call() throws Exception{
    Random generator = new Random();
    Integer randomNumber = generator.nextInt(5);
    Thread.sleep(randomNumber * 1000);
    return randomNumber;
  }
}
```

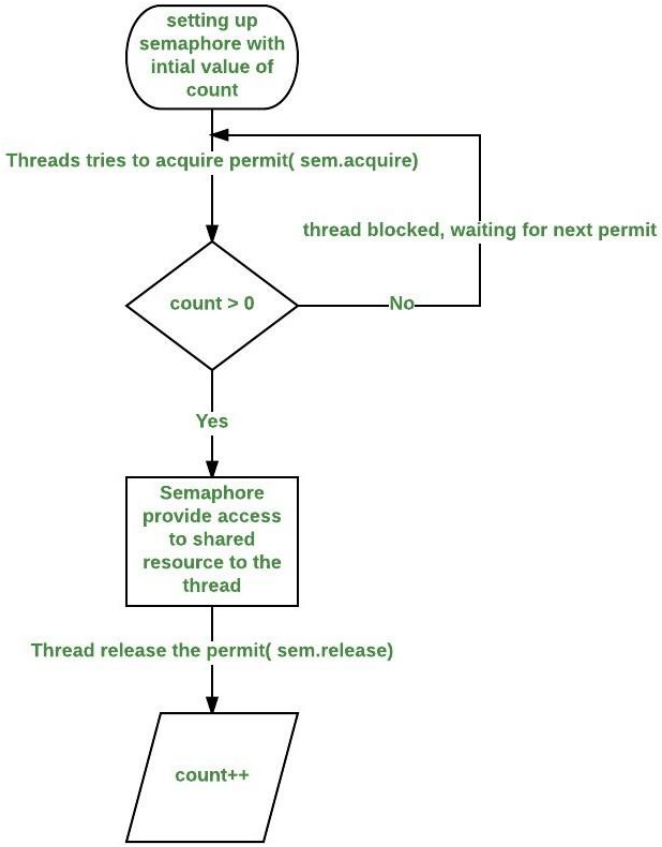
14.2. Future

- When the call() method completes, answer must be stored in an object known to the main thread, so that the main thread can know about the result that the thread returned. For this, a Future object can be used.
- 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.
- The Future can be made to work with Runnable as well.
- Some important methods of future:
    - **public boolean cancel(boolean mayInterrupt):** Used to stop the task. It stops the task if it has not started. If it has started, it interrupts the task only if mayInterrupt is true.
    - **public Object get() throws InterruptedException, ExecutionException:** Used to get the result of the task. If the task is complete, it returns the result immediately, otherwise it waits till the task is complete and then returns the result.
    - **public boolean isDone():** Returns true if the task is complete and false otherwise.

```
ExecutorService executor = Executors.newSingleThreadExecutor();
Future<Long> result = executor.submit(new CallableExample());
System.out.println("Result = " + result.get());
```

15. Semaphore in Java

- A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.
- Java provide **Semaphore** class in `java.util.concurrent` package that implements this mechanism.



- Constructors:
  - **Semaphore(int permits):**
    Creates a Semaphore with the given number of permits and nonfair fairness setting.
  - **Semaphore(int permits, boolean fair):**
    Creates a Semaphore with the given number of permits and the given fairness setting.

- Important methods:
  - **void acquire()** :
    This method acquires a permit, if one is available and returns immediately, reducing the number of available permits by one. If the current thread is interrupted while waiting for a permit then InterruptedException is thrown.
  - **void acquire(int permits)** :
    This method acquires the given number of permits, if they are available, and returns immediately, reducing the number of available permits by the given amount.If the current thread is interrupted while waiting for a permit then InterruptedException is thrown.

- **boolean tryAcquire()**:
  This method acquires a permit, if one is available and returns immediately, with the value true, reducing the number of available permits by one. If no permit is available then this method will return immediately with the value false.
- **void release()**:
  This method releases a permit, increasing the number of available permits by one. If any threads are trying to acquire a permit, then one is selected and given the permit that was just released.
- **void release(int permits)**:
  This method releases the given number of permits, increasing the number of available permits by that amount.If any threads are trying to acquire permits, then one is selected and given the permits that were just released. If the number of available permits satisfies that thread's request then that thread is (re)enabled for thread scheduling purposes; otherwise the thread will wait until sufficient permits are available.
- **int availablePermits()**:
  This method returns the current number of permits available in this semaphore.This method is typically used for debugging and testing purposes.
- **int drainPermits()**:
  This method acquires and returns all permits that are immediately available.

```
Semaphore sem = new Semaphore(1);
sem.acquire();
sem.release();
```

15.1. Fairness

- The constructor for Semaphore optionally accepts a *fairness* parameter.
- When set false, this class makes no guarantees about the order in which threads acquire permits.
- In particular, *barging* is permitted, that is, a thread invoking `acquire()` can be allocated a permit ahead of a thread that has been waiting - logically the new thread places itself at the head of the queue of waiting threads.
- When fairness is set true, the semaphore guarantees that threads invoking any of the `acquire` methods are selected to obtain permits in the order in which their invocation of those methods was processed (first-in-first-out; FIFO).
- Note that FIFO ordering necessarily applies to specific internal points of execution within these methods. So, it is possible for one thread to invoke `acquire` before another, but reach the ordering point after the other, and similarly upon return from the method.
- Also note that the untimed `tryAcquire` methods do not honor the fairness setting, but will take any permits that are available.

16. CountDownLatch

- CountDownLatch is used to make sure that a task waits for other threads before it starts. To understand its application, let us consider a server where the main task can only start when all the required services have started.
- When we create an object of CountDownLatch, we specify the number of threads it should wait for, all such thread are required to do count down by calling countDown() once they are completed or ready to the job. As soon as count reaches zero, the waiting task starts running.
- This is a one-shot phenomenon - the count cannot be reset.

- Constructor of CountDownLatch:
  - **CountDownLatch(int count):**
    Constructs a CountDownLatch initialized with given count.

- Methods of CountDownLatch:
  - **void await():**
    Causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted. Any subsequent invocations of await return immediately.
  - **Boolean await(long timeout, TimeUnit unit):**
    Causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted, or the specified waiting time elapses.
  - **void countDown():**
    Decrements the count of the latch, releasing all waiting threads if the count reaches zero.
  - **long getCount():**
    Returns the current count.

```
CountDownLatch latch = new CountDownLatch(1);

new Thread(() -> {
        latch.countDown();
}).start();

latch.await();
```

17. CyclicBarrier

- A **CyclicBarrier** is a synchronizer that allows a set of threads to wait for each other to reach a common execution point, also called a **barrier**.
- 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.

- Constructors:
  - **CyclicBarrier(int parties)**
    Creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and does not perform a predefined action when the barrier is tripped.
  - **CyclicBarrier(int parties, Runnable barrierAction)**
    Creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and which will execute the given barrier action when the barrier is tripped, performed by the last thread entering the barrier.

- Important Methods:
  - **int await():**
    Waits until all parties have invoked await on this barrier.
  - **int await(long timeout, TimeUnit unit):**
    Waits until all parties have invoked await on this barrier, or the specified waiting time elapses.
  - **int getNumberWaiting():**
    Returns the number of parties currently waiting at the barrier.
  - **int getParties():**
    Returns the number of parties required to trip this barrier.
  - **boolean isBroken():**
    Queries if this barrier is in a broken state.
  - **void reset():**
    Resets the barrier to its initial state.

```
CyclicBarrier barrier = new CyclicBarrier(1);
new Thread(() -> {
        try {
            barrier.await();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }).start();
```

18. Producer Consumer Problem

19. BlockingQueue

20. SynchronousQueue

- A type of blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa.
- A synchronous queue does not have any internal capacity, not even a capacity of one.
- You cannot peek at a synchronous queue because an element is only present when you try to remove it; you cannot insert an element (using any method) unless another thread is trying to remove it; you cannot iterate as there is nothing to iterate.
- The head of the queue is the element that the first queued inserting thread is trying to add to the queue; if there is no such queued thread then no element is available for removal and poll() will return null.
- For purposes of other Collection methods (for example contains), a SynchronousQueue acts as an empty collection.
- This queue does not permit null elements.
- They are well suited for handoff designs, in which an object running in one thread must sync up with an object running in another thread in order to hand it some information, event, or task.
- This class supports an optional fairness policy for ordering waiting producer and consumer threads. By default, this ordering is not guaranteed. However, a queue constructed with fairness set to true grants threads access in FIFO order.

- Constructors:
  - **SynchronousQueue<E>():**
    Creates a SynchronousQueue with nonfair access policy.
  - **SynchronousQueue<E>(boolean fair):**
    Creates a SynchronousQueue with the specified fairness policy.

- Important Methods:
  - **void put(E o)**
    Adds the specified element to this queue, waiting if necessary for another thread to receive it.
  - **E take()**
    Retrieves and removes the head of this queue, waiting if necessary for another thread to insert it.

```
SynchronousQueue<Integer> queue = new SynchronousQueue<>();

Runnable producer = () -> {
    Integer producedElement = ThreadLocalRandom.current().nextInt();
    queue.put(producedElement);
};

Runnable consumer = () -> {
        Integer consumedElement = queue.take();
};
```

21. Exchanger

- The Exchanger class in Java can be used to share objects between two threads of type T. The class provides only a single overloaded method exchange(T t).
- When invoked exchange waits for the other thread in the pair to call it as well. At this point, the second thread finds the first thread is waiting with its object. The thread exchanges the objects they are holding and signals the exchange, and now they can return.
- This pattern of exchanging data while reusing the buffer allows having less garbage collection.

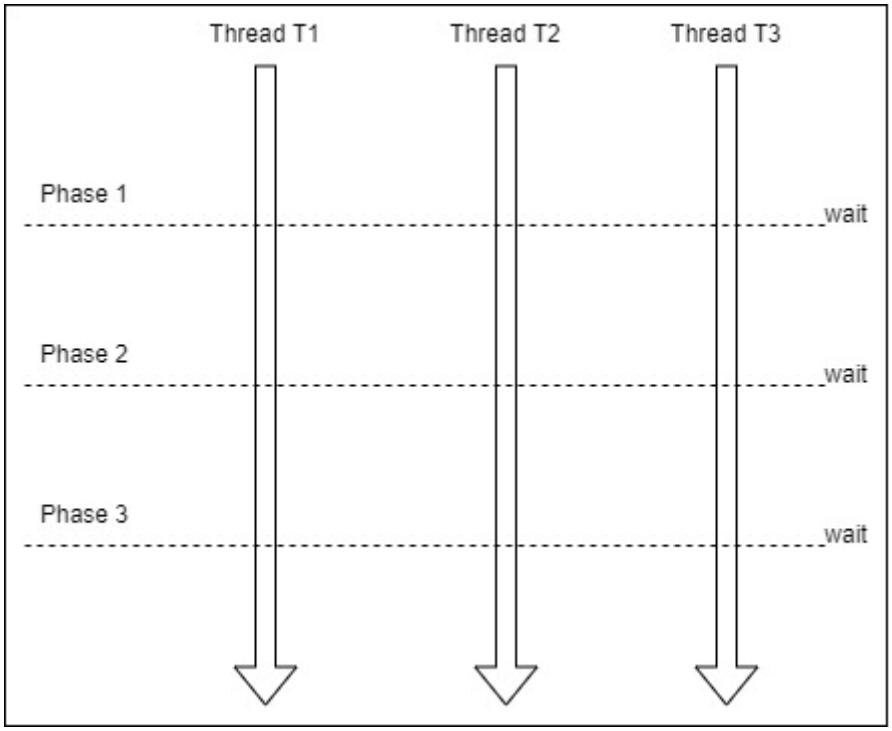
- Constructor:
  - **Exchanger<V>():**
    Creates a new Exchanger.

- Methods:
  - **V exchange(V v)**
    Waits for another thread to arrive at this exchange point (unless the current thread is interrupted), and then transfers the given object to it, receiving its object in return.
  - **V exchange(V v, long timeout, TimeUnit unit)**
    Waits for another thread to arrive at this exchange point (unless the current thread is interrupted or the specified waiting time elapses), and then transfers the given object to it, receiving its object in return.

```
Exchanger<String> exchanger = new Exchanger<>();

Runnable runnable = () -> {
         String randomString = UUID.randomUUID().toString();
         randomString = exchanger.exchange(randomString);
   };

new Thread(runnable, "T1").start();
new Thread(runnable, "T2").start();
```

22. Phaser

- Phaser's primary purpose is to enable synchronization of threads that represent one or more phases of activity.
- It lets us define a synchronization object that waits until a specific phase has been completed. It then advances to the next phase until that phase concludes.
- It can also be used to synchronize a single phase, and in that regard, it acts much like a CyclicBarrier.



- Constructors:
  - **Phaser():**
    Creates a new phaser with no initially registered parties, no parent, and initial phase number 0.
  - **Phaser(int parties):**
    Creates a new phaser with the given number of registered unarrived parties, no parent, and initial phase number 0.
  - **Phaser(Phaser parent):**
    Equivalent to Phaser(parent, 0).
  - **Phaser(Phaser parent, int parties):**
    Creates a new phaser with the given parent and number of registered unarrived parties.

- Methods:
  - **int arrive():**
    Arrives at this phaser, without waiting for others to arrive.
  - **int arriveAndAwaitAdvance():**
    Arrives at this phaser and awaits others.
  - **int arriveAndDeregister():**
    Arrives at this phaser and deregisters from it without waiting for others to arrive.
  - **int awaitAdvance(int phase):**

Awaits the phase of this phaser to advance from the given phase value, returning immediately if the current phase is not equal to the given phase value or this phaser is terminated.

- **int awaitAdvanceInterruptibly(int phase):**
  Awaits the phase of this phaser to advance from the given phase value, throwing InterruptedException if interrupted while waiting, or returning immediately if the current phase is not equal to the given phase value or this phaser is terminated.
- **int awaitAdvanceInterruptibly(int phase, long timeout, TimeUnit unit):**
  Awaits the phase of this phaser to advance from the given phase value or the given timeout to elapse, throwing InterruptedException if interrupted while waiting, or returning immediately if the current phase is not equal to the given phase value or this phaser is terminated.
- **int bulkRegister(int parties):**
  Adds the given number of new unarrived parties to this phaser.
- **void forceTermination():**
  Forces this phaser to enter termination state.
- **int getArrivedParties():**
  Returns the number of registered parties that have arrived at the current phase of this phaser.
- **Phaser getParent():**
  Returns the parent of this phaser, or null if none.
- **int getPhase():**
  Returns the current phase number.
- **int getRegisteredParties():**
  Returns the number of parties registered at this phaser.
- **Phaser getRoot():**
  Returns the root ancestor of this phaser, which is the same as this phaser if it has no parent.
- **int getUnarrivedParties():**
  Returns the number of registered parties that have not yet arrived at the current phase of this phaser.
- **Boolean isTerminated():**
  Returns true if this phaser has been terminated.
- **protected boolean onAdvance(int phase, int registeredParties):**
  Overridable method to perform an action upon impending phase advance, and to control termination.
- **int register():**
  Adds a new unarrived party to this phaser.

```
public class PhaserDemo {
    public static void main(String[] args) {
        Phaser phaser = new Phaser();
        phaser.register();
        int currentPhase;

        new TestingThread(phaser, "A");
        new TestingThread(phaser, "B");

        currentPhase = phaser.getPhase();
        phaser.arriveAndAwaitAdvance();
        System.out.println("Phase " + currentPhase + " Complete");
        System.out.println("Phase Zero Ended");
```

```
            currentPhase = phaser.getPhase();
            phaser.arriveAndAwaitAdvance();
            System.out.println("Phase " + currentPhase + " Complete");
            System.out.println("Phase One Ended");

            currentPhase = phaser.getPhase();
            phaser.arriveAndAwaitAdvance();
            System.out.println("Phase " + currentPhase + " Complete");
            System.out.println("Phase Two Ended");

            phaser.arriveAndDeregister();
            if (phaser.isTerminated()) {
                System.out.println("Phaser is terminated");
            }
        }
    }

    class TestingThread implements Runnable {
        private final Phaser phaser;
        private final String title;

        public TestingThread(Phaser phaser, String title) {
            this.title = title;
            this.phaser = phaser;
            phaser.register();
            new Thread(this).start();
        }

        @Override
        public void run() {
         try {
            System.out.println("Thread: " + title + " Phase 0 Started");
            phaser.arriveAndAwaitAdvance();

            Thread.sleep(10);

            System.out.println("Thread: " + title + " Phase 1 Started");
            phaser.arriveAndAwaitAdvance();

            Thread.sleep(10);

            System.out.println("Thread: " + title + " Phase 2 Started");
            phaser.arriveAndDeregister();
         } catch (InterruptedException e) {
            System.out.println(e);
         }
    }
}
```