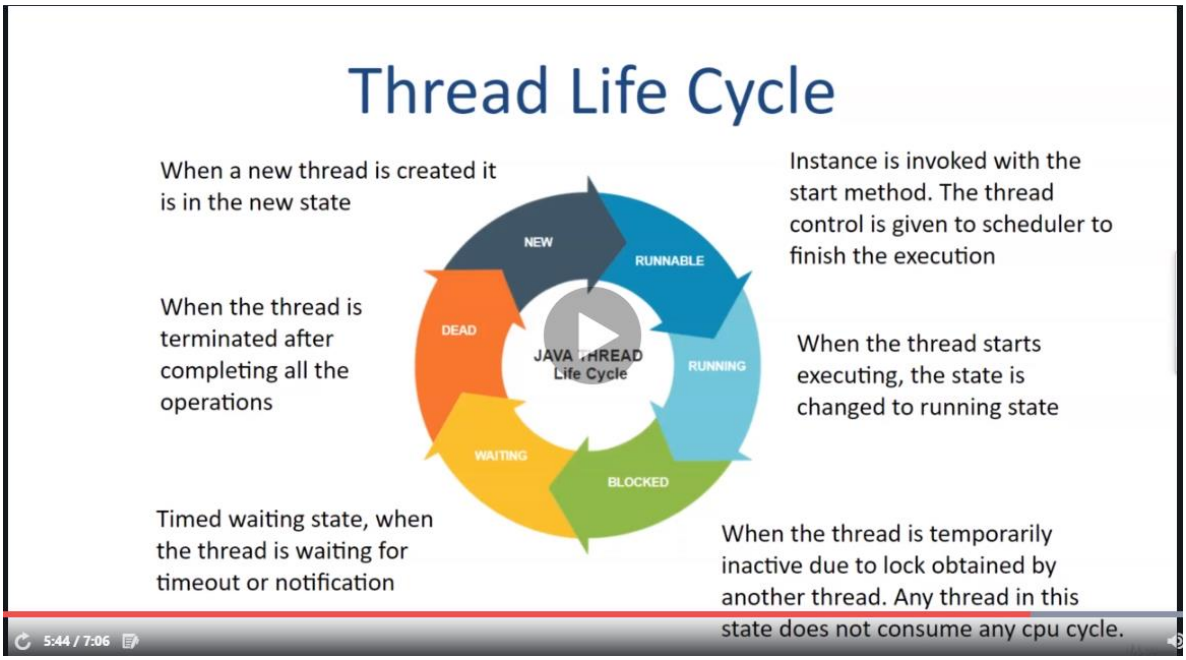
Thread Life Cycle

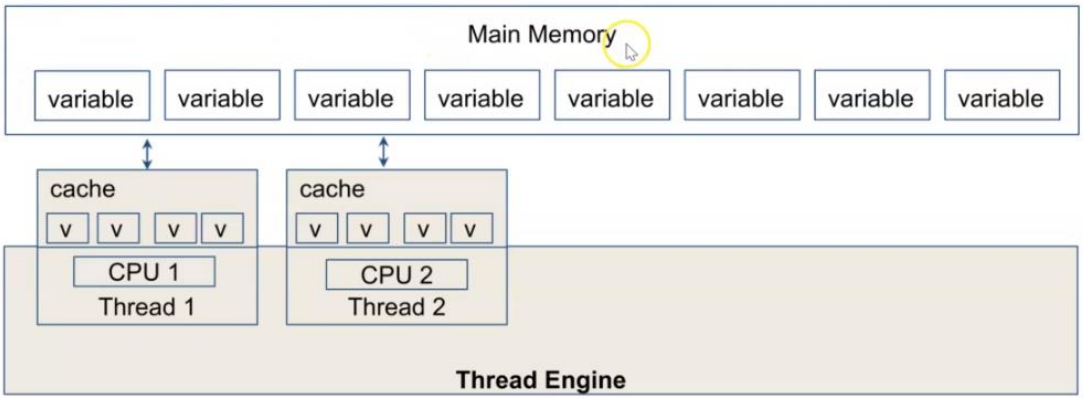


Join

1. The join method allows one thread to wait for the completion of another thread
2. When we invoke the join() on a thread, the calling thread goes into a waiting state. It remains in the waiting state until the referenced thread terminates
3. The join method will keep waiting if the referenced thread is blocked which can become an issue as the calling thread will become non-responsive. Java provided two overloaded version of the join() method that takes timeout period
4. Join throws InterruptedException it the referenced thread is interrupted
5. If the referenced thread unable to start or already terminated join will return immediately

Volatile

Volatile keyword in Java is used as an indicator to Java compiler and Thread that do not cache value of this variable and always read it from main memory



Every read of volatile variable is from RAM, don't make a variable volatile if you don't have to. Cache is

Volatile

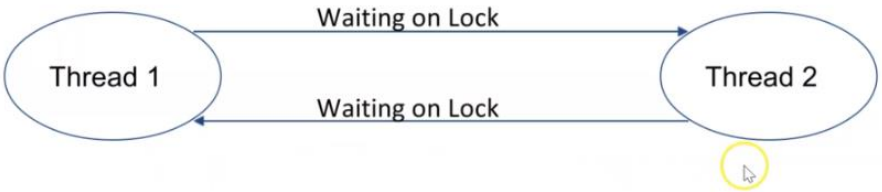
1. Variable can be defined as volatile, methods and classes cannot be volatile
2. Volatile keywords guarantee that the variable will be read from main memory (RAM) and not from thread local cache
3. In Java reads and writes are atomic for all variables declared volatile
4. Java volatile keyword doesn't mean atomic operation, it's a common misconception that declaring variable volatile will make all operation on the variable atomic. You still need to ensure exclusive access using synchronized method or block
5. If a variable is not shared between threads, don't use volatile
6. Every read of volatile variable is from RAM, don't make a variable volatile if you don't have to. Cache is much faster than RAM

Volatile

In the scenario where only one thread is writing to a variable and other thread is just reading (like in case of status flag, we will show code example for this next) volatile helps in the correct visibility of the value of the variable. But volatile is not enough if many threads are reading and writing the value of the shared variable. In that case because of race condition threads may still get wrong value.

Deadlock

Deadlock is a situation in which threads are waiting for each other to release the lock. There may be situations where a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since both threads are waiting for each other to release the lock the condition is called deadlock.



Bank Deposit Deadlock Example

In this example, two different threads attempts to transfer money between the two same accounts at the same time.

Here is what happens:

1. Student A is transferring his semester fee ($3000) to the University bank account
2. University is refunding ($1000) for the classes Student A didn't take in the previous semester
3. Both Student and University are trying to transfer money to each other account at the same time

Bank Deposit Deadlock Example

Implementation:

1. Thread 1 is transferring the money from student account to university bank account. There is no lock on the student account so a lock is granted to Thread1
2. Thread 2 is transferring the money from University account to student bank account. There is no lock on the university account so a lock is granted to Thread2
3. Thread 1 is now trying to get a lock on university account as it already has a lock on student account in order to transfer the money from student account to university account
4. Thread 2 is also trying to get a lock on student account as it already has a lock on university account in order to transfer the money from university account to student account
5. Thread 1 is waiting for Thread2 to release the lock
6. Thread 2 is waiting for Thread1 to release the lock
7. Threads are in a deadlock state as each thread will be waiting indefinitely for each other to release the lock

Livelock

Livelock is like deadlock in the sense that two (or more) processes are blocking each other but with livelock, each process is actively trying to resolve the problem on its own (like reverting back and retry). A livelock happens when the combination of these processes efforts to resolve the problem make it impossible for them to ever terminate.

An example of livelock is; a husband and wife are trying to eat soup, but only have one spoon between them. Each spouse is too polite, and will pass the spoon if the other has not yet eaten.

Another example of livelock is; two people attempting to pass each other in a corridor:
PersonA moves to his left to let PersonB pass, while PersonB moves to his right to let PersonA pass. Seeing that they are still blocking each other, PersonA moves to his right, while PersonB moves to his left. They're still blocking each other.

Bank Deposit Livelock Example

Implementation:

1. Transaction 1 withdraw $3000 from Student A account
2. Transaction 2 withdraw $1000 from University account
3. Transaction 1 failed to deposit money into University Account because Transaction 2 has a lock on University account. Student account is refunded
4. Transaction 2 failed to deposit money into Student Account because Transaction 1 has a lock on Student account. University account is refunded
5. Transaction 1 again withdraw $3000 from Student A account
6. Transaction 2 again withdraw $1000 from University Account
7. Transaction 1 again failed to deposit money into University Account because Transaction 2 has a lock on University account. Student account is refunded
8. Transaction 2 again failed to deposit money into Student Account because Transaction 1 has a lock on Student account. University account is refunded

And so on.....

Synchronization

What is Java Synchronization

Java synchronization provides capabilities to control the access of multiple threads to any shared resource

Why use Synchronization

1. To Prevent interference from multiple threads
2. To avoid consistency issues

Thread Synchronization

There are 2 types of thread synchronization

1. Mutual Exclusive
   a. Synchronized Method
   b. Synchronization block
   c. Static synchronization
2. Cooperation (Inter-thread communication)
   a. Wait ()
   b. Notify()
   c. NotifyAll()

Mutual Exclusive

Synchronization is a modifier in java which is used for mutual exclusive lock. With the help of synchronization modifier we can restrict a shared resource to be accessed by only one thread. There are three ways to use synchronization modifier:

1. By synchronized method
2. By synchronized block
3. By static synchronization

If an object or a block is declared as synchronized than only one thread can access that object or block at a time. No other thread can take the object or block until it is available.

Synchronization is built around an internal entity known as the intrinsic lock or monitor lock. Every object has an intrinsic lock associated with it. A thread that need exclusive and consistent access to an object's field has to acquire the object intrinsic lock before accessing them and then release the intrinsic lock when it's done with them.

Intrinsic Lock

Every object in JVM has internally a lock. Synchronized keyword tries to acquire that lock of the target object

Synchronized Method Intrinsic Lock

Synchronized instance methods: The implicit lock is 'this', which is the object used to invoke the method. Each instance of this class will use their own lock. Example: A new instance of object has 3 methods - MethodA & Method B are synchronized and MethodC is not synchronized

Synchronized static methods: The lock is the Class object. All instances of this class will use the same lock. Example: a method MethodA is defined as synchronized static.

Synchronized Block Intrinsic Lock

The implicit lock can be instance or class

Wait, Notify & NotifyAll

Wait, Notify and Notifyall are used for threads communication via a common object, this common object is considered a medium for inter thread communication via these methods.
These methods need to be called from synchronized context, otherwise an exception IllegalMonitorStateException is thrown.

wait(): When you call wait on the object it tells the thread to give up the lock and go to sleep state until some other threads enters in same monitor and calls notify or notify all.

notify(): When you call notify on the object it wakes one thread waiting for the object. If multiple threads are waiting for the object it will call one of them. Depending on the OS implementation it will call one of the waiting thread.

notifyAll(): notifyAll will wake up all threads waiting on the object. Which thread will be called first is depending on the thread priority and OS implementation.

Locks

A lock is a thread synchronization mechanism like synchronized blocks except locks can be more sophisticated than Java synchronized blocks.

Since lock is an interface, we need to use one of its implementation to use locks in applications. ReentrantLock is one such implementation of lock interface.

Difference between Lock Interface and synchronized keyword

The main difference between a lock and synchronized block are:

1. Synchronized block does not provide a timeout. Using Lock.tryLock(long timeout, TimeUnit timeUnit) it is possible
2. The synchronized block must be fully implemented in a single method. A lock can have its call to lock and unlock in separate methods.

Reentrant Lock

- Reentrant lock allow thread to enter into lock on a resource more than once.
- When the thread first enters into lock a hold count is set to one.
- Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one.
- For every unlock request hold count is decremented by one and when hold count is 0, the resource is unlocked.

Reentrant locks also offer a fairness parameter, by which the lock would abide by the order of the lock request. When the thread unlocks the resource the lock would go to the thread which has been waiting for the longest time. This fairness mode is setup by passing true to the constructor of the lock.

Reentrant Lock

Reentrant lock methods:

lock(): call to the lock() method increments the hold count by 1 and gives the lock if the shared resources is free

unlock(): call to the unlock() method decrements the hold count by 1, when this count is reached to zero the resource is released

trylock(): if the shared resource is available the trylock will lock the resource and increment the hold count to 1. If the shared resource is not available it will exit instead of waiting

trylock(long timeout, TimeUnit unit): the thread wait for certain amount of time specified as a parameter before exiting if the shared resource is not available

lockInterruptibly(): If a thread is waiting for a lock but some other thread request the lock, then the current thread will be interrupted and return immediately without acquiring lock

Semaphore

Semaphore is a class in java.util.concurrent package.

Semaphores are generally used to restrict the number of threads to access resources.

Semaphore basically maintains a set of permits through the use of counter.

To access the resource, a thread must be granted a permit from the semaphore.

If the counter is greater than zero, then access is allowed. If it is zero, then access is denied.

Semaphore

1. If the semaphore count is greater than zero, then the thread acquires a permit, which causes the semaphore's count to be decremented
2. Otherwise the thread is block until a permit can be acquired
3. A thread releases the permit when the thread no longer need an access to the shared resource and thereby increasing the counter
4. If a thread is waiting for the permit it will acquire the permit and continue its execution

Examples of Semaphores:

1. Restricting number of connections to the database
2. Bounding a collection

Executor

Managing and executing a few threads is easy but it becomes difficult when we have hundreds of thousands of threads running simultaneously.

Executor framework help you in creating and managing threads in an application.

The most important feature of this framework is the separation of concerns. It lets the developers create tasks (Runnable and Callable) and let the framework decide when, how and where to execute that task on a Thread which is totally configurable

Executor

How does Executor work

- The work of an executor is to execute tasks.
- The executor picks up a thread from the threadpool to execute a task.
- If a thread is not available and new threads cannot be created then the executor stores these tasks in a queue. A task can also be removed from the queue

ThreadPoolExecutor

- Executor is the base interface of the executor framework and has only one method execute(Runnable Command).
- ExecutorService and ScheduledExecutorService are two other interface which extends executor.
- ThreadPoolExecutor is an implementation of the ExecutorService interface. The ThreadPoolExecutor executes the given tasks using one of its internally pool threads.

Thread pool executors

There are five types of thread pool executors with pre-build methods in Executors interface

1. **Fixed thread pool executor:** Creates a thread pool of fixed number of threads. Any task over the number of threads will wait in the queue until the thread is available

```
ThreadPoolExecutor executor = Executors.newFixedThreadPool(5);
```

2. **Cached thread pool executor:** Creates a thread pool that creates new threads as needed. It will reuse the previously constructed threads when they are available. Use this thread pool with caution as this can bring down system if the number of threads goes beyond what the system can handle

```
ThreadPoolExecutor executor = Executors.newCachedThreadPool();
```

Thread pool executors

3. **Scheduled thread pool executor:** Creates a thread pool that can schedule commands to run after a given delay or to execute periodically
ThreadPoolExecutor executor = Executors.newScheduledThreadPool(10);

4. **Single thread pool executor:** Creates single thread to execute all tasks, use it when you have one task to execute
ThreadPoolExecutor executor = Executors.newSingleThreadExecutor();

5. **Work stealing thread pool executor:** Creates a thread pool that maintains enough threads to support the given parallelism level. Here parallelism level means the maximum number of threads which will be used to execute a given task, at a single point of time, in multiprocessor machines
ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newWorkStealingPool(5);

How to delegate tasks to an ExecutorService

There are a few different ways to delegate tasks to an ExecutorService.

1. **execute (Runnable)** : It takes Runnable implementation and execute it asynchronously
2. **submit (Runnable)** : It takes Runnable implementation and returns a future object The future object can be used to check if the Runnable has finished executing
3. **Submit (Callable)** : It takes Callable implementation and returns a future object
4. **invokeAny()** : It takes collection of callable objects. Invoking this method does not return any future but returns the result of one of the callable objects
5. **invokeAll()** : This method invokes all callable objects passed as parameters. It returns the future objects which can be used to get the results of the execution of each Callable

Callable & Future

- Callable interface represents a thread that can return a value. It is very much similar to Runnable interface except that it can return a value.
- Callable interface can be used to compute status or results that can be returned to invoking thread.

Future

When the call() method completes, return value must be stored in an object in main thread, so that the main thread can know about the result that the thread returned. For this, a Future object can be used. It holds Future as an object as a 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

---

Callable & Future

Methods of Future

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.

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.

boolean isDone(): Returns true if the task is complete and false otherwise.

Callable & Future Example

1. In one of the previous sections we implemented a sum of 5000 numbers in parallel using runnable interface. In this section we will be implementing the same example using callable interface.
2. To accomplish the sum in parallel we will be creating a new class SumOfNumbersUsingCallable. Inside the class we will creating two callable threads, first one will sum the number from 0-2500 and other will sum from 2501 - 5000. We will run these callable interface using executorservice
3. We will use IntStream to create a list of 5000 numbers
4. We will use executor invokeAll to get the results of individual callable and sum them and print the total sum

CountDownLatch

CountDownLatch is a synchronization aid which allows one Thread to wait for one or more Threads before starts processing. Simply put, a CountDownLatch has a counter field, which you decrement as required and use it to block a calling thread until it's counted down to zero.

While doing parallel processing we will initiate the CountDownLatch with the same value of the counter as the number of threads we want it to wait for. Then, we will call await() method in the main thread and call the countdown() to decrease the counter inside each thread the main thread waiting for to complete.

CountDownLatch

Methods of CountDownLatch

void await(): Waits until the latch has counted down to zero

boolean await(long timeout, TimeUnit unit): similar to wait but waits only for given time and returns false when not reached zero

void countDown(): Decrements the number in the latch. Notifies awaiting threads when reached zero

long getCount(): Returns current value of latch

CountDownLatch Example

1. We will be implementing the sum of 5000 numbers with CountDownLatch
2. To accomplish the sum in parallel we will be creating a new class SumOfNumbersUsingLatch. Inside the class we will be creating two callable threads, first one will sum the number from 0-2500 and other will sum from 2501 - 5000. We will create a countDownLatch instance with counter of 2. We will call the countDown() at the end of the callable threads to decrease the counter.
3. We will use IntStream to create a list of 5000 numbers
4. We will use ExecutorService and will call the callable using submit() to get the results in Future object and sum the future objects once the await() returns

CyclicBarrier

CyclicBarrier is synchronisation aid that allows a set of threads to wait for each other to reach a common barrier point. The threads who reached the barrier point has to wait for other threads to reach. As soon as all the threads have reached the barrier point, all of them are released to continue.

A CyclicBarrier is reusable, so it's more like a racing tour where everyone meets at a waypoint before proceeding on the next leg of the tour. Cyclicbarrier can be reused after the waiting threads are released and that is where it is different than CountdownLatch. We can reuse CyclicBarrier by calling reset() method which resets the barrier to its initial state.

CyclicBarrier

Example of CyclicBarrier: A multithreaded download manager. The download manager will start multiple threads to download each part of the file simultaneously. Suppose that you want the intergity check for the downloaded pieces to be done after a particular time interval. Here cyclicbarrier plays an important role. After each time interval, each thread will wait at the barrier so that thread associated with cyclicbarrier can do the integrity check. This integrity check can be done multiple times thanks to CyclicBarrier

CyclicBarrier Example

1. We will be implementing the sum of 15000 numbers with CyclicBarrier in three steps
2. In the first step we will sum numbers from 0-5000 using 2 thread
3. In the second step we will sum numbers from 5001 - 10000 using 2 threads
4. In the third step we will sum numbers from 10001 - 15000 using 2 threads
5. At the end of each step all the thread will wait for each other to complete and then proceed to the next step using CyclicBarrier awaits() method
6. We print the total sum after the 3 steps
7. We will use ExecutorService and will call the callable using submit() to get the results in Future object and sum the future objects once the await() returns

BlockingQueue

The Java BlockingQueue interface, java.util.concurrent.BlockingQueue, represents a queue which is thread safe to put elements into, and take elements out of from. In other words, multiple threads can be inserting (queue) and taking elements (dequeue) concurrently from a Java BlockingQueue, without any concurrency issues arising.

A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. If a thread tries to take an element and there are none left in the queue, the thread can be blocked until there is an element to take and similarly the thread is blocked if the queue is full and it tries to add an element to the queue. Whether or not the calling thread is blocked depends on what methods you call on the BlockingQueue.

BlockingQueue Methods

BlockingQueue methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future. It is not possible to insert null into a BlockingQueue. If you try to insert null, the BlockingQueue will throw a NullPointerException.

	Throws Exception	Special Value	Blocks	Times Out
Insert	add(o)	offer(o)	put(o)	offer(o,timeout, timeunit)
Remove	remove(o)	poll()	take()	poll(timeout, timeunit)
Examine	element()	peek()		

1. **Throws Exception**: If the attempted operation is not possible immediately, an exception is thrown.
2. **Special Value**:If the attempted operation is not possible immediately, a special value is returned (often true / false).
3. **Blocks**:If the attempted operation is not possible immediately, the method call blocks until it is.
4. **Times Out**:If the attempted operation is not possible immediately, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).

BlockingQueue Implementation

BlockingQueue is an interface so you need to use one of its implementation. The java.util.concurrent package has the following implementation of BlockingQueue:

1. **ArrayBlockingQueue**: ArrayBlockingQueue class is backed by an array and it is a bounded blocking queue. By bounded it means that the size of the Queue is fixed. Once created, the capacity cannot be changed
2. **DelayQueue**: DelayQueue is a specialized Priority Queue that orders elements based on their delay time. It means that only those elements can be taken from the queue whose time has expired
3. **LinkedBlockingQueue:** LinkedBlockingQueue is an optionally-bounded blocking queue based on linked nodes. It means that the LinkedBlockingQueue can be bounded, if its capacity is given, else the LinkedBlockingQueue will be unbounded
4. **PriorityBlockingQueue**: PriorityBlockingQueue is an unbounded blocking queue that uses the same ordering rules as class PriorityQueue and supplies blocking retrieval operations
5. **SynchronousQueue**: SynchronousQueue is special kind of BlockingQueue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa

ArrayBlockingQueue

- ArrayBlockingQueue class is one of the implementation of BlockingQueue.
- ArrayBlockingQueue is an array backed bounded queue. By bounded it means that the size of the queue is fixed. Once created the capacity of queue cannot be changed.
- Attempts to put an element into a full queue will result in the operation blocking. Similarly attempts to take an element from an empty queue will also be blocked.
- Boundness of the ArrayBlockingQueue can be achieved initially by passing capacity as the parameter in the constructor of ArrayBlockingQueue.

ArrayBlockingQueue

- This queue orders elements FIFO (first-in-first-out). The head of the queue is that element that has been on the queue the longest time. The tail of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.
- ArrayBlockingQueue supports an optional fairness policy for ordering waiting producer and consumer threads. With fairness set to true, the queue grants threads access in FIFO order.

ArrayBlockingQueue Example

1. We will be implementing a producer consumer example with ArrayBlockingQueue
2. We will be creating two Runnables one for producer and other for consumer
3. Producer will be putting numbers to the ArrayBlockingQueue using put()
4. Consumer will be taking numbers from the ArrayBlockingQueue using take()
5. Both put() and take() are blocking so if the queue is empty the thread will be blocked and similarly if the queue is full put() will be blocked

DelayQueue

- DelayQueue class is an unbounded blocking queue of delayed elements, in which an element can only be taken when its delay has expired.
- DelayQueue class is part of java.util.concurrent package.
- The head of the queue is that Delayed element whose delay expired furthest in the past.
- If no delay has expired there is no head and poll will return null.
- Expiration occurs when an element's getDelay(TimeUnit.NANOSECONDS) method returns a value less than or equal to zero. Even though unexpired elements cannot be removed using take or poll, they are otherwise treated as normal elements. For example, the size method returns the count of both expired and unexpired elements. This queue does not permit null elements.

DelayQueue

What is delay element?
A delay element implements java.util.concurrent.Delayed interface and it's getDelay() method return a zero or negative value which indicate that the delay has already elapsed

For clarity, we can consider that each element stores it activation date/time. As soon as, this timestamp reaches, element is ready to be picked up from queue.

An implementation of Delayed interface must define a compareTo() method that provides an ordering consistent with its **getDelay()** method.

compareTo(Delayed o) method does not return the actual timestamp, generally. It return a value less than zero if the object that is executing the method has a delay smaller than the object passed as a parameter – otherwise a positive value greater than zero. It will return zero if both the objects have the same delay.

DelayQueue Example

1. We will be implementing a producer consumer example with Delay Queue
2. We will create a DelayTask class which implements Delayed interface
3. We will override getDelay() and compareTo()
4. We will be creating two Runnables one for producer and other for consumer
5. Producer will be putting DelayTask objects to the DelayQueue using put() and with a Random expiration time
6. Consumer will be consuming DelayTask from the DelayQueue using take()

Linked Blocking Queue

- Linked blocking queue is based on linked nodes
- This queue orders elements FIFO (first-in-first-out)
- The head of the element is that element that has been on the queue the longest time
- The tail of the queue is that element that has been on the queue the shortest time
- New elements are inserted at the tail of the queue
- The queue retrieval operation is from the head of the queue

PriorityBlockingQueue

Java PriorityBlockingQueue class is a concurrent blocking queue data structure implementation in which objects are processed based on their priority. The "blocking" part of the name is added to imply the thread will block until there's an item available on the queue.

In a priority blocking queue, the elements are ordered as per their natural ordering or based on a custom Comparator supplied at the time of creation.

PriorityBlockingQueue

PriorityBlockingQueue Features:

1. A priority queue is unbounded, but has an internal capacity governing the size of an array used to store the elements on the queue. It is always at least as large as the queue size. As elements are added to a priority queue, its capacity grows automatically. The default initial capacity is '11' which can be overridden using initialCapacity parameter in appropriate constructor
2. The objects of the priority queue are ordered by default in natural order
3. It does not permit null elements
4. It does not permit insertion of non-comparable objects
5. PriorityBlockingQueue is thread safe
6. PriorityBlockingQueue class and its iterator implements all of the optional methods of the Collection and Iterator interfaces

PriorityBlockingQueue Example

1. We will be implementing a producer consumer example with PriorityBlockingQueue
2. We will be creating two Runnables one for producer and other for consumer
3. We will be creating an array of Strings (names) and producer will put() them in the queue
4. Consumer will be taking names from the queue as per their natural ordering using take()
5. Both put() and take() are blocking so if the queue is empty the thread will be blocked and similarly if the queue is full put() will be blocked

SynchronousQueue

- Synchronous queue is 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.
- Synchronous queues are basically used for handoff purposes.

SynchronousQueue Example

1. We will be implementing a producer consumer example with SynchronousQueue
2. We will be creating two Runnables one for producer and other for consumer
3. We will be creating an array of Strings (names) and producer will put() them in the queue once consumer is available to take them one by one
4. put() must wait for a corresponding take() operation
5. We will be running this using executorservice

Difference b/w Traditional & Concurrent Collections

- Most of the classes in the traditional collection like ArrayList, LinkedList, HashSet, LinkedHashset, TreeSet, HashMap are non-synchronized and therefore they are not thread-safe.
- All the classes present in Concurrent Collection are synchronized in nature and are thread-safe
- Traditional classes have some classes like Vector, Hashtable and Stack, which are synchronized in nature and are Thread Safe
- Traditional classes have SynchronizedCollection, SynchronizedSet, SynchronizedList and SynchronizedMap methods through which we can get synchronized version of non-synchronized objects

Difference b/w Traditional & Concurrent Collections

- Synchronized collections are much slower than their concurrent counterparts. Main reason of this slowness is locking, synchronized collection locks whole object, it means when a thread is iterating over elements in a collection, all other collection methods blocks, causing other threads having to wait. Other threads cannot perform other operations on the collection until the first thread release the lock. This causes overhead and reduces performance
- Concurrent collection achieves thread-safety smartly e.g. ConcurrentHashMap divides the whole map into several segments and locks only on segments, which allows multiple threads to access other collections without locking.

Difference b/w Traditional & Concurrent Collections

- CopyOnWriteArrayList allows multiple reader thread to read without synchronization and when a write happens it copies the ArrayList and swaps. So if we use concurrent collection classes in their favorable conditions e.g. for more reading and less updates, they are much more scalable than synchronized collections.
- Traditional collection throws ConcurrentModificationException if a thread is iterating a collection and if another thread try to add new element in that iterating object simultaneously. Concurrent collection does not throw this exception
- Traditional collection are good choice for single threaded applications where as concurrent collection are good choice for multi-threaded applications

Example

1. We will be implementing a example to show the difference b/w traditional and concurrent collection
2. Step1 : We will be adding elements to an ArrayList while iterating and the program should throw ConcurrentModificationException
3. Step 2: We will also be adding elements to a concurrent list while iterating with CopyOnWriteArrayList which is a concurrent collection

ConcurrentHashMap

- ConcurrentHashMap class is introduced in JDK1.5, which implements ConcurrentMap as well as Serializable interface.
- Prior to Java1.5 if you need a Map implementation, which is thread safe and can be safely used in a concurrent and multi-threaded environment, then you only have Hashtable or synchronized Map because hashmap is not thread-safe.
- ConcurrentHashMap provides better performance than HashTable and Synchronized Map because it only locks a portion of Map, instead of whole Map, which is the case with Hashtable and synchronized Map.
- ConcurrentHashMap allows concurrent read operations and at the same time maintains integrity by synchronizing write operations.

ConcurrentHashMap

Key points of ConcurrentHashMap:

- The underlined data structure for ConcurrentHashMap is Hashtable.
- ConcurrentHashMap class is thread-safe i.e. multiple thread can operate on a single object without any complications.
- In ConcurrentHashMap, the Object is divided into number of segments according to the concurrency level. Default concurrency-level of ConcurrentHashMap is 16.
- ConcurrentHashMap allows multiple readers to read concurrently without any blocking but for updation in object, thread must lock the particular segment in which thread want to operate. This type of locking mechanism is known as Segment locking or bucket locking. Hence at a time 16 updation operations can be performed by threads.

Example

1. We will be implementing a token generation example using ConcurrentHashMap
2. We will be creating two threads which will be inserting tokens in ConcurrentHashMap simultaneously
3. We will be creating a third thread which will be reading tokens from the ConcurrentHashMap while other threads are creating tokens

NavigableMap

A ConcurrentMap supporting NavigableMap operations, and recursively so for its navigable sub-maps.

Navigablemap supports concurrent access, and has concurrent access enabled for its submaps. The "submaps" are the maps returned by various methods like headMap(), subMap() and tailMap().

Important Navigable Maps methods:

headMap()
Returns a view of the portion of this map whose keys are strictly less than toKey

tailMap()
Returns a view of the portion of this map whose keys are greater than or equal to fromKey

subMap()
Returns a view of the portion of this map whose keys range from fromKey to toKey.

Example

1. We will be implementing a NavigableMap example using ConcurrentSkipListMap. ConcurrentSkipList map is an implementation of NavigableMap. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.
2. We will be implementing headMap(), tailMap() and subMap() methods in this example

Fork-Join

With the advancement in the hardware where even mobile phones have multiple cores the programming paradigm requires effective use of multiple cpus/cores. The effective use of parallel cores in a Java application has always been a challenge. Java 7 has therefore implemented a Fork and Join framework that would distribute the work across multiple cores then join them to return the result set.

Fork-Join breaks the task at hand into mini-tasks until the mini-tasks is simple enough that it can be solved without further breakups. Fork-Join is like divide and conquer algorithm. The fork-join framework is distinct because it uses a work-stealing algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy. Fork-Join framework is based on Doug Lea's work on java concurrency and it employs the following pseudocode

```
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
 }}
```

Fork-Join

Fork : A task that uses fork and split itself into small subtasks

Join : A task that joins all the results into one result

ForkJoinPool

ForkJoinPool is a specialized implementation of ExecutorService implementation with the work-stealing algorithm. If you recall we submit multiple independent calls to ExecutorService which are then executed by the thread pool threads. To execute a big task we have to perform following action:

1. Split the task into small sub tasks
2. Process sub tasks independently
3. Join the sub tasks results into a final result

Fork Join internally does all the above steps for us. We just submit a single task to ForkJoinPool and ForkJoinPool will work to divide and conquer the problem and return the result

ForkJoinPool

ForkJoinPool Creation

We create a ForkJoinPool using its constructor, the constructor takes the level of parallelism you desire. Parallelism dictates the number of threads or cpus you want to work concurrently to work on the task

Submitting Tasks to the ForkJoinPool

You can submit two types of tasks to the ForkJoinPool - **RecursiveAction** and **RecursiveTask**. The only difference between these two classes is that the RecursiveAction does not return a value while RecursiveTask does have a return value and returns an object of specified type.

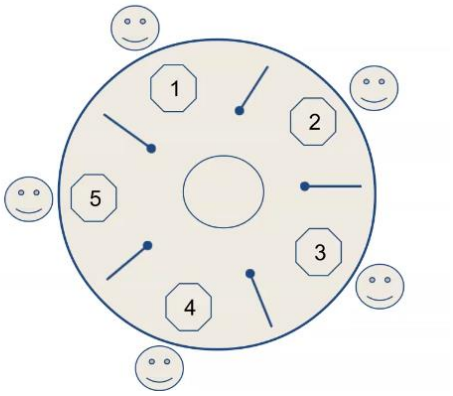
Dining Philosophers

Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their right or the one on their left as they become available, but cannot start eating before getting both forks.

Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed.

Design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve; *i.e.*, each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.



Implementation

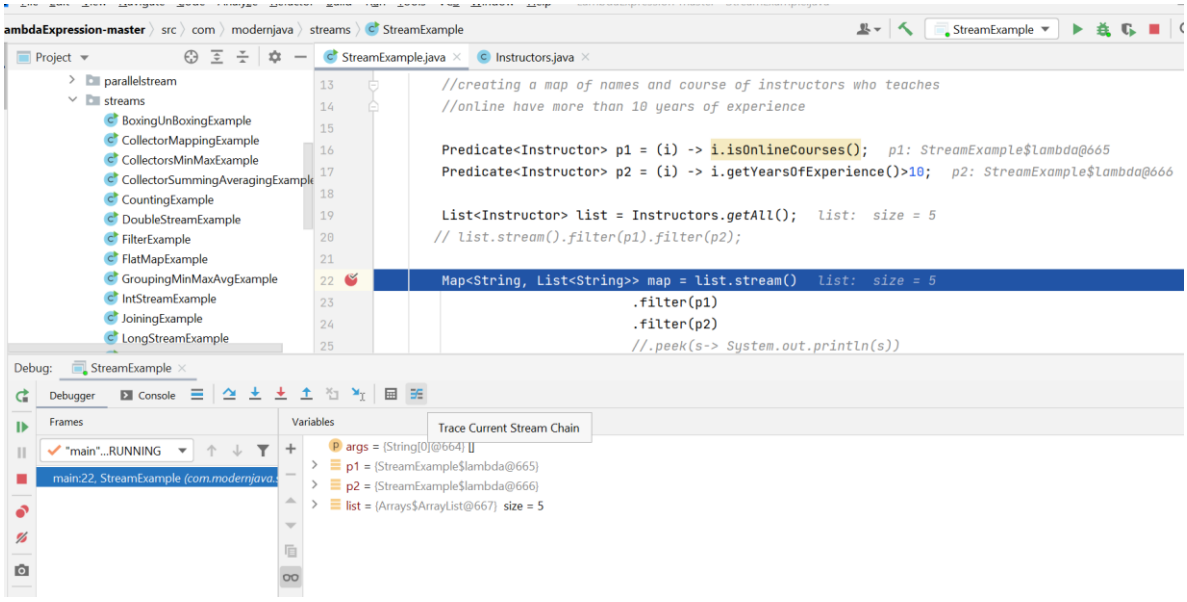
1. We will be implementing following pseudocode using lock and synchronized

```
While (true) {
    think();
    Pick_up_left_fork;
    Pick_up_right_fork;
    eat();
    Put_down_right_fork;
    Put_down_left_fork;
}
```

Debug Stream in IntelliJ:

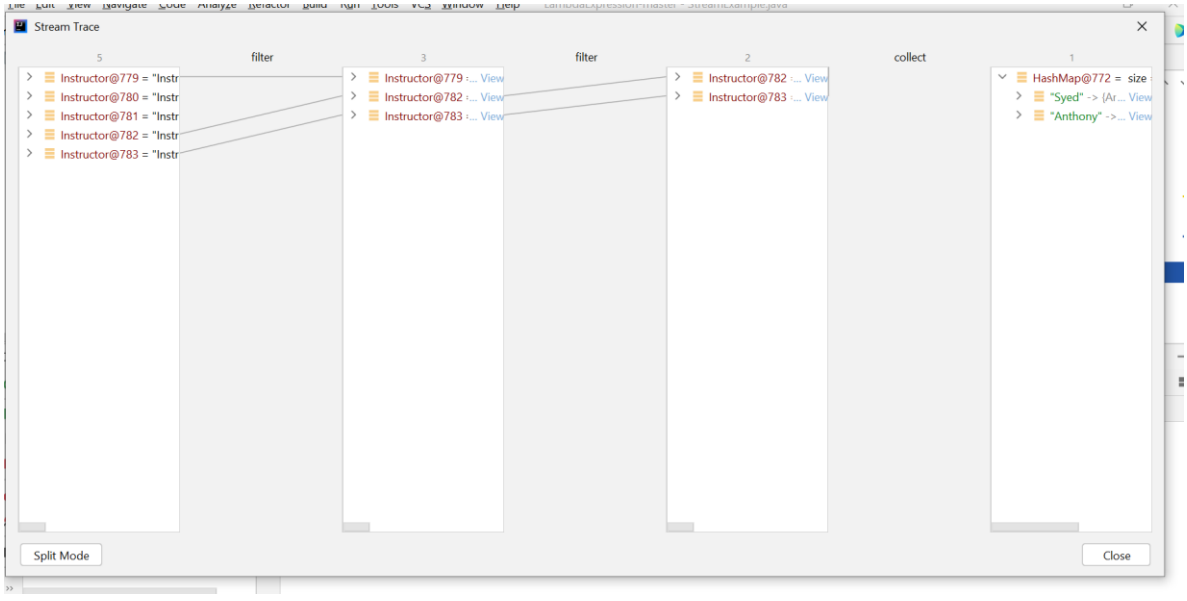
```
//creating a map of names and course of instructors who teaches
 //online have more than 10 years of experience

 Predicate<Instructor> p1 = (i) -> i.isOnlineCourses();
 Predicate<Instructor> p2 = (i) -> i.getYearsOfExperience()>10;

 List<Instructor> list = Instructors.getAll();
// list.stream().filter(p1).filter(p2);

 Map<String, List<String>> map = list.stream()
                         .filter(p1)
                         .filter(p2)
                         //.peek(s-> System.out.println(s))
                         .collect(Collectors.toMap(Instructor::ge
tName, Instructor::getCourses));

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

Step1 : Put Debug point on stream then start execution in debug mode

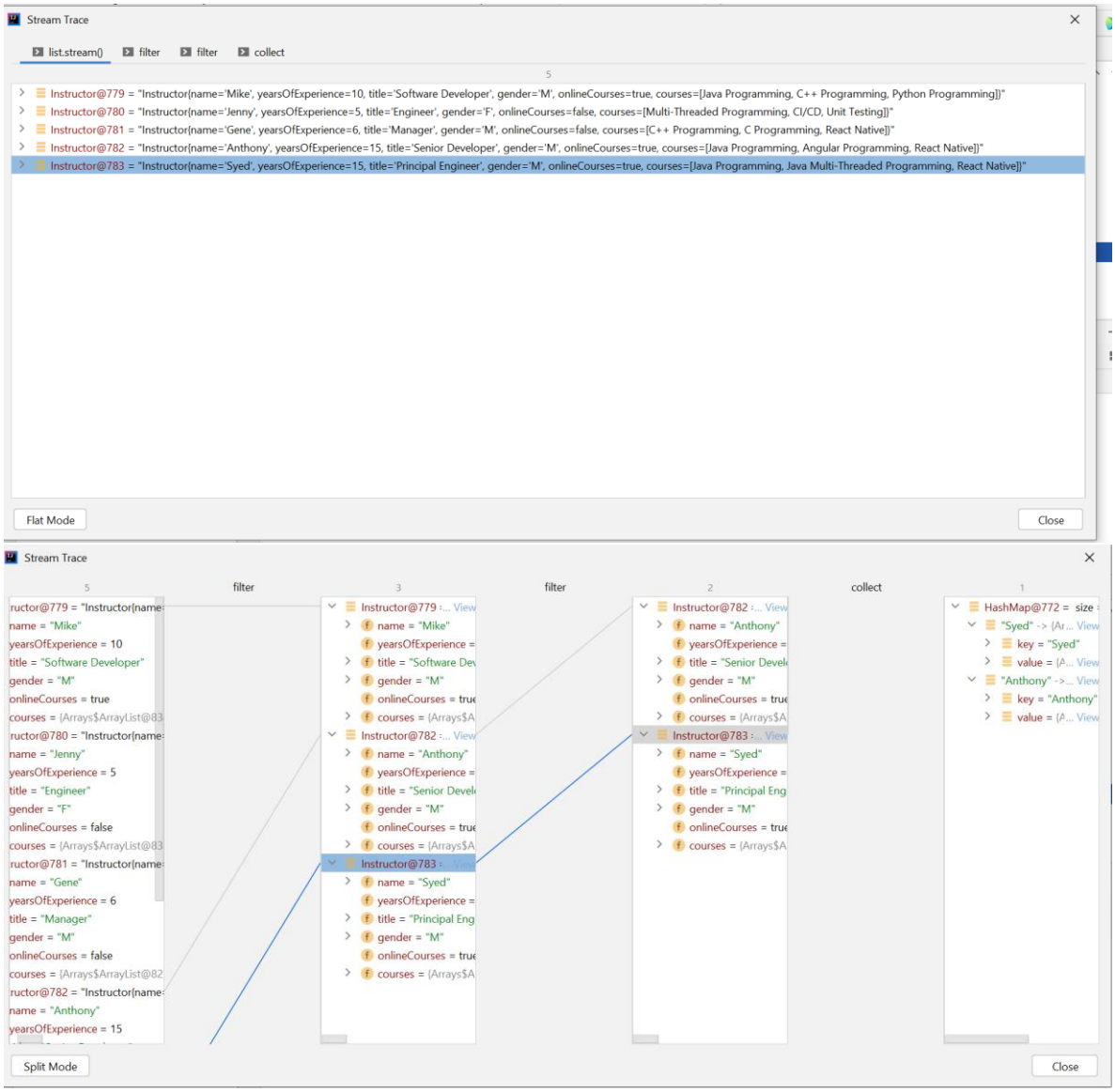


Click on trace current stream in bottom you can in below screen each operation:

First mode: flat mode



Split mode:



Another Mehod for debug by stream API which is peek ():

```
Map<String, List<String>> map = list.stream()
                    .peek(s-> System.out.println(s))
                        .filter(p1)
                        .filter(p2)
                        .peek(s-> System.out.println( "after
filter :"+s))
                        .collect(Collectors.toMap(Instructor::get
Name, Instructor::getCourses));
```