**Stopping Thread in the middle: stop()** method of the Thread class could be used to stop the thread in the middle. But this is the **dangerous** thing to do as it leaves the system in inconsistent state, because we are not giving the opportunity to the thread to rollback or reverse the actions that it has taken. And hence the stop method is deprecated and should not be used.

Correct approach would be to call the **interrupt()** method on the thread and then it is up to the thread to consider whether to stop or not.

A thread can then check if it was interrupted or not using **interrupted()** method of the Thread class. We can design the thread in a way that it reverses the actions/operations performed and then stop when interrupted.

Note - If you are not extending Thread class and instead implementing Runnable interface, then you can use **Thread.isInterrupted()** to check if the current thread is interrupted.

interrupted() and Thread.isInterrupted() both methods return true if the thread is interrupted when it is alive.

1. class MyThread extends Thread {
3. public void run() {
5. // Intentionally kept in infinite loop
6. for( ;; ) {
8. // Returns true if the thread is interrupted.
9. if (interrupted()) {
11. // You are supposed to rollback or reverse the operation
12. // in progress and stop.
14. System.out.println("Thread is interrupted hence stopping..");
16. // Terminates the loop.
17. break;
18. }
20. System.out.print("T");
21. }
22. }
23. }
25. public class Main {
27. public static void main(String[] args) {
29. MyThread thr = new MyThread();
30. thr.start();
32. // Just for demo, wait for 2 seconds
33. // before interrupting thr.
34. try {
35. Thread.sleep(2000);
36. } catch (InterruptedException e) {
37. e.printStackTrace();
38. }
40. // Interrupt the thread.
41. thr.interrupt();
42. }
43. }

**sleep() method -**

sleep() method of the thread class is used to block the thread for the given time interval in milliseconds. This method throws InterruptedException if the thread is interrupted while it is in sleep.

**Thread States:** A thread can be in one of the following states:

**NEW**

A thread that is created but not yet started is in this state.

**RUNNABLE**

A thread executing in the Java virtual machine is in this state, internally we can think of it as a combination of two sub states Ready and Running, i.e. when you start the thread it comes to Ready state and wait for the CPU, and if CPU is allocated then it goes into Running state. Once allocated CPU time is completed, in other words when the Thread schedular suspends the thread then it goes back to the Ready state and waits for its turn again.

The yield() method instructs the thread schedular to pass the CPU to other waiting thread if any.

**BLOCKED**

A thread is blocked if it is waiting for a monitor lock is in this state. Refer **synchronized** methods and blocks.

**WAITING**

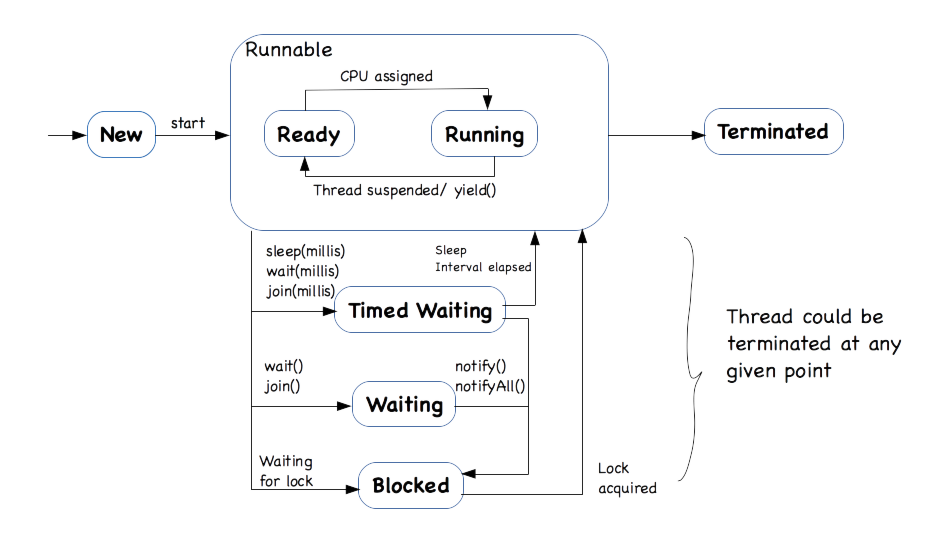
A thread that is waiting indefinitely for another thread to perform a particular action is in this state. Refer wait()**,**join()

**TIMED\_WAITING**

A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state. Refer wait(millis), join(millis)

**TERMINATED**

A thread that has existed i.e. it has either completed its task or terminated in the middle of the execution.



yield() method -

yield() method is important in few scenarios, suppose a thread is given 5 min of CPU time, now after a minute thread knows that it doesn't need the CPU anymore with in that time period, in such scenarios do you think that blocking the CPU for the next four minutes is a good idea ? No, it is better to pass on the control to the threads if any waiting for CPU and that is when we can use the yield() method. Usage Thread.yield(), it is a static method of the Thread class and it affects the current thread from which the method is invoked.

**Thread Priorities**

Thread priorities range between 1 and 10.

**MIN\_PRIORITY** - 1 being the minimum priority

**NORM\_PRIORITY** - 5 is the normally priority, this is the default priority value.

**MAX\_PRIORITY** - 10 being the max priority.

setPriority(int newPriority)  -

A method in the Thread class, this is used to set the new priority for the thread. If the *newPriority* value is more than the maximum priority allowed for the group then maximum priority is considered, i.e. if you try to set 15 then it takes only 10. And for a given ThreadGroup if the maximum allowed priority is 7 then any thread within that group can have a maximum of 7.

Just think about a software installer app, the thread that copies the files should be given more priority than the thread which display the progress etc, that speeds up the installation process. Below example demonstrates setting higher priority for copyThread.

1. class CopyTask implements Runnable {
2. @Override
3. public void run() {
4. while(true) {
5. System.out.print("C");
6. }
7. }
8. }
10. class ProgressTask implements Runnable {
11. @Override
12. public void run() {
13. while(true) {
14. System.out.print("-");
15. }
16. }
17. }
19. public class Main {
21. public static void main(String[] args) {
22. CopyTask copyTask = new CopyTask();
23. Thread copyThread = new Thread(copyTask);
24. copyThread.setPriority(Thread.NORM\_PRIORITY + 3);
25. copyThread.start();
27. ProgressTask progressTask = new ProgressTask();
28. Thread progressThread = new Thread(progressTask);
29. progressThread.start();
30. }
31. }

**Internal System Threads and ThreadGroup**

**Thread.currentThread() -**

currentThread() is a static method in the class Thread and all the static method in the Thread class normally operate on the thread in which it is being executed. Here Thread.currentThread() returns a reference to the current thread i.e. the main thread.

**getThreadGroup() - [Thread class method]**

A Thread class method that returns a reference to the ThreadGroup to which the corresponding thread instance belongs.

**getParent() - [ThreadGroup class method]**

A ThreadGroup class method that returns a reference to the parent thread group if any. If there is no parent then this method returns null.

**setMaxPriority(int maxPriority) - [ThreadGroup class method]**

Sets the maximum priority for that group so that no thread can exceed this priority with in the group.

Example -

1. public class Main {
3. public static void main(String[] args) {
4. System.out.println("System threads..........");
5. Thread thr = Thread.currentThread();
6. ThreadGroup grp = thr.getThreadGroup();
7. while (grp.getParent() != null) {
8. grp = grp.getParent();
9. }
10. grp.list();
11. }
12. }

Sample Output -

**NOTE - OUTPUT MAY VARY WITH JAVA VERSION.**

1. System threads..........
2. java.lang.ThreadGroup[name=system,maxpri=10]
3. Thread[Reference Handler,10,system]
4. Thread[Finalizer,8,system]
5. Thread[Signal Dispatcher,9,system]
6. java.lang.ThreadGroup[name=main,maxpri=10]
7. Thread[main,5,main]

**Associating a Thread with ThreadGroup -**

Straight forward, create an instance of the ThreadGroup and give it a name. You can do group level operations over the ThreadGroup object. To associate a thread with the thread group, pass the thread group reference to the Thread class constructor.

Example -

1. class MyTask implements Runnable {
3. @Override
4. public void run() {
5. try {
6. Thread.sleep(5000);
7. } catch (InterruptedException e) {
8. e.printStackTrace();
9. }
10. }
11. }
12. public class Main {
14. public static void main(String[] args) {
16. // CREATING A THREADGROUP
17. ThreadGroup myGroup = new ThreadGroup("MyGroup");
18. myGroup.setMaxPriority(4);
20. // ASSOCIATING A THREAD WITH THREAD GROUP
21. Thread myThread = new Thread(myGroup, new MyTask(), "DemoThread");
22. myThread.start();
24. System.out.println("System threads..........");
25. Thread thr = Thread.currentThread();
26. ThreadGroup grp = thr.getThreadGroup();
27. while (grp.getParent() != null) {
28. grp = grp.getParent();
29. }
30. grp.list();
31. }
32. }

Sample Output -

1. System threads..........
2. java.lang.ThreadGroup[name=system,maxpri=10]
3. Thread[Reference Handler,10,system]
4. Thread[Finalizer,8,system]
5. Thread[Signal Dispatcher,9,system]
6. java.lang.ThreadGroup[name=main,maxpri=10]
7. Thread[main,5,main]
8. java.lang.ThreadGroup[name=MyGroup,maxpri=4]
9. Thread[DemoThread,4,MyGroup]

Technical Note -

It is important to note that even the main method runs with a thread called the main thread. And its default priority is 5.

**Daemon Threads**

Daemon threads are the ones which does not prevent the JVM from exiting the application once it finishes. Daemon threads are handy for performing background tasks such as garbage collection or collecting application statistics etc. Note that the Java Virtual Machine exits when the only threads running are all daemon threads.

Example -

In the below example at the end of the while loop grp will point to system thread group. And enumerate method lists the threads in that group and copies the references to the given array. It also returns the number of threads copied. And later I am printing the thread name along with the boolean value checking if it is a daemon thread or not, using the method isDaemon().

1. public class Main1 {
3. public static void main(String[] args) {
5. System.out.println("System threads..........");
7. Thread thr = Thread.currentThread();
8. ThreadGroup grp = thr.getThreadGroup();
9. while (grp.getParent() != null) {
10. grp = grp.getParent();
11. }
13. Thread [] threads = new Thread[10];
14. int n = grp.enumerate(threads);
16. for (int i=0; i < n; i++) {
17. System.out.println(
18. "Thread Name: " + threads[i].getName() +
19. " ; isDaemon: " + threads[i].isDaemon());
20. }
21. }
22. }

Output -

1. System threads..........
2. Thread Name: Reference Handler ; isDaemon: true
3. Thread Name: Finalizer ; isDaemon: true
4. Thread Name: Signal Dispatcher ; isDaemon: true
5. Thread Name: main ; isDaemon: false

You can see that Reference Handler, Finalizer, Single Dispatcher all these are daemon threads because they run the background and they doesn't stop the application from exiting.

setDaemon(boolean on) -

The method of the Thread class makes a enables us to set whether a thread is a daemon thread or user thread.

Example -

1. class MyTask implements Runnable {
3. @Override
4. public void run() {
5. for (;;) {
6. System.out.print("T");
7. }
8. }
9. }
10. public class Main {
11. public static void main(String[] args) {
12. Thread thr = new Thread(new MyTask());
13. thr.setDaemon(true);
14. thr.start();
16. for (int i=1; i <= 200; i++) {
17. System.out.print(" M ");
18. }
19. }
20. }

Output -

MMTTMMT

A combination of M and T but the application ends once the main ends.

**Callable Task**

Callable interface -

Unlike Runnable, Callable interface allows us to create an asynchronous task which is capable of returning an Object.

1. interface Callable<V> {
2. V call() throws Exception;
3. }

If you implement Runnable interface we can not return any result. But if you implement Callable interface then you can return the result as well. Like run() method in the Runnable interface, you need to override the call() method. The return type of the call() method should match with the intended return type of the result. Callable<Double> means the call method returns Double value,  Callable<Fruit> means call method returns an instance of type Fruit.

1. class GetStockQuoteTask implements Callable<Double> {
3. private String stockSymbol;
5. public GetStockQuoteTask(String stockSymbol) {
6. this.stockSymbol = stockSymbol;
7. }
9. public Double call() {
10. // Write some logic to fetch the stock price
11. // for the given symbol and return it.
12. return 0.0;
13. }
14. }

To submit this task for execution, you can use the submit method on the ExecutorService.

1. String symbol = "ABCD";
2. GetStockQuoteTask task = new GetStockQuoteTask(symbol);
3. Future<Double> future = executor.submit( task );
4. Double price = future.get();

Future Object -

When you submit a Callable task to the ExecutorService, it returns a Future object. This object enables us to access the request and check for the result of the operation if it is completed.

**Important methods -**

**isDone()**- Returns true if the task is done and false otherwise.

**get()** - Returns the result if the task is done, otherwise waits till the task is done and then it returns the result.

**cancel(boolean mayInterrupt)** - Used to stop the task, stops it immediately if not started, otherwise interrupts the task only if mayInterrupt is true.

Example -

1. import java.util.concurrent.Callable;
2. import java.util.concurrent.ExecutorService;
3. import java.util.concurrent.Executors;
4. import java.util.concurrent.Future;
6. class MyMath {
7. public static int add(int a, int b) {
8. return a + b;
9. }
10. }
12. public class Main {
14. public static void main(String[] args) throws Exception {
16. int x = 10;
17. int y = 20;
19. ExecutorService executor =
20. Executors.newFixedThreadPool(1);
22. // Submit a Callable task and use the Future
23. // object to fetch the result.
24. Future<Integer> future =
25. executor.submit(
26. new Callable<Integer>() {
27. public Integer call() {
28. return MyMath.add(x, y);
29. }
30. });
32. // do some parallel task
33. // Inefficient to simply wait,
34. // instead you can release the CPU
35. // by calling Thread.yield() inside
36. // the while loop.
37. while( ! future.isDone())
38. ; // wait
40. // fetch the result
41. int z = future.get();
43. System.out.println( "Result is " + z );
44. }
45. }

**Thread Synchronization**

Thread Synchronization

  Thread synchronization is used to solve concurrency problems that exist in parallel processing. Concurrency problem exist when more than one thread is accessing the same shared resource.

E.g.

1) More than one transaction is being performed on the same account

2) Multiple resources are booking tickets for the same train from different locations.

etc.

Synchronization in Java Threads

It can be achieved through

  1) Synchronized methods

  2) Synchronized block

  3) Locks discussed later.

synchronized method

Consider the class Sample

1. class Sample {
2. synchronized void f() {...}
3. }

Consider Three threads T1, T2, T3 and two objects for Sample they are A,B.

T1  ..........A.f();  locks A and proceeds

T2  ..........B.f();  locks B and proceeds

T3  ..........A.f();  wait till T1 unlocks A.

To run a synchronized method object must be locked.

synchronized block

When synchronization is not required for the entire method i.e only certain part of the code must be synchronized then we use synchronized block.

1. synchronized( object ) {
2. // operations over the object
3. }

The above code is executed only after obtaining lock over the object.

Thread Safe Code or Re-entrant code:-

When a code block is safe from concurrency problems then the code is referred as thread safe or re-entrant.In case of the below operation**incr()** operation is considered as thread safe or re-entrant.

Example -

In the below example try removing synchronized keyword before the incr() operation and check the result. You will find inconsistent result. By making the method **synchronized**, we are forcing the thread to lock the object before performing the**incr()** operation. Though control is intentionally passed to other thread, other thread won't be able to proceed with the operation as it need to first lock (obj) before proceeding forward.

i.e. lets assume **t1** locks **obj** then **t2** should wait till **t1** releases the lock, hence object is modified by only one thread at a time and you will see consistent results.

1. class Sample {
3. private int x;
5. public int getX() {
6. return x;
7. }
9. public void setX(int x) {
10. this.x = x;
11. }
13. /\*
14. \* Try removing synchronized.
15. \*/
16. public synchronized void incr() {
18. // read the value of x.
19. int y = getX();
21. // Increment the value
22. y++;
24. // Just assume if control is switched to
25. // some other thread and it too looks at
26. // the old value of x and proceeds with
27. // modification. Such scenarios lead to
28. // in consistent results.
29. // To simulate such scenarios lets us just
30. // pass the control to some other thread.
32. // with sleep this thread will go to blocked state
33. // for the given time interval, hence other thread
34. // will get a chance.
35. try { Thread.sleep(1); } catch(Exception e) {}
37. // set x to new value.
38. setX(y);
39. }
40. }
42. class MyThread extends Thread {
44. Sample obj;
46. public MyThread(Sample obj) {
47. this.obj = obj;
48. }
50. public void run() {
51. obj.incr();
52. }
54. }

57. public class Main {
59. public static void main(String[] args) {
61. Sample obj = new Sample();
62. obj.setX(10);
64. // In this case both the threads t1 and t2
65. // are sharing the same Sample object obj.
66. // Both the threads will try to perform the
67. // increment operation simultaneously.
69. MyThread t1 = new MyThread(obj);
70. MyThread t2 = new MyThread(obj);
72. t1.start();
73. t2.start();
75. // Here main thread called the join operation
76. // on t1 and t2. join() operations waits for
77. // thread to complete before returning.
78. try {
79. t1.join();
80. t2.join();
81. } catch (InterruptedException e) {
82. e.printStackTrace();
83. }
85. System.out.println( obj.getX() );
86. }
88. }

Synchronizing static data operations -

Is the below increment() operation thread-safe ?

1. class Sample {
3. static int a = 5;
4. int b = 10;
6. public synchronized void increment( ) {
7. a++;
8. b++;
9. }
11. // ....
12. }

Answer is NO. Because we made it synchronized we may think that it is thread-safe but it is not. And it is because of the static variable a. Lets assume that there are two objects for Sample, in that case both of them will share the same copy of a because it is a class member, where as they get different copies of b, because b is non static i.e. the instance member and each instance will get a separate copy of b.

Assume Thread 1 invoked the increment method over the first object and Thread 2 invoked the increment method over the second object. Because the increment() method is non-static and it is synchronized, object should be locked before getting into the method.

Here Thread1 locks the first object and gets in and also Thread 2 locks the second object and gets in, because both are different objects and hence both the thread acquire locks and they both proceed with the operation.

You can see that b++ is not having any issues, because both the threads are operating on different copies of b, but what about a++ it is still not thread safe.

Solution 1 -

1. class Sample {
3. static int a = 5;
4. int b = 10;
6. public  void increment( ) {
7. // lock the Class object before modifying
8. // static content.
9. synchronized(Sample.class) {
10. a++;
11. }
13. // lock the object before modifying
14. // instance members.
15. synchronized(this) {
16. b++;
17. }
18. }
20. // ....
21. }

In this case the increment() operation is thread-safe, because for modifying the static member 'a' we are locking the class object, Sample.class returns a reference to class object and synchronized block will acquire lock over the object and then proceed forward with the operation. And for b++ we are locking the current object using the this reference. And hence both the operations are now thread-safe, as we properly locked the corresponding object before modification.

Solution 2 -

1. class Sample {
3. static int a = 5;
4. int b = 10;
6. // This method is static and hence it locks the Class object.
7. public static synchronized void incrementA( ) {
8. a++;
9. }
11. // This method is non static and hence it locks the object
12. // on which it is invoked.
13. public synchronized void increment( ) {
14. incrementA();
15. b++;
16. }
18. // ....
20. }

Create a static method for incrementing a and declare it as synchronized, so that when the thread enters this method it locks the class object. Just remember that whenever you are modifying the static content in a multi-threaded environment you should lock the class object to make your code thread-safe.

Issue with synchronized methods -

Synchronized methods doesn't always solve concurrency problems. Lets consider a simple List class and assume that the size() and add() operations are synchronized

1. class List {
2. ...
3. public synchronized int size(){
4. ...
5. }
6. public synchronized void add(Object value) {
7. ....
8. }
9. }

You can think that there is no concurrency issue here. But, lets analyze a simple scenario consider that list should not contain more than 5 elements, and assume that list is already having 4 elements and two threads are trying to insert an element into the list.

**Thread 1 -**

1. 1) if (list.size() < 5) {
2. 2) list.add(value1);
3. }

**Thread 2 -**

1. a) if (list.size() < 5) {
2. b) list.add(value2);
3. }

Lets assume this execution sequence **(1)(a)(2)(b)** in this case both threads will see that list size is 4 and is less than 5. Hence both will add an element into the list, which makes the list size as 6 violating the condition. You can see the issue is not resolved even with both **size()** and **add()** being synchronized methods.

Solution -

You should apply thread synchronization at operation level with the help of synchronized block. i.e.

**Thread 1 -**

1. 1) synhronized(list) {
2. 2) if (list.size() < 5) {
3. 3) list.add(value1);
4. }
5. }

**Thread 2 -**

1. a) synchronized(list) {
2. b) if (list.size() < 5) {
3. c) list.add(value2);
4. }
5. }

Now consider the execution sequence**(1)(a)(2)(3)(b)(c)**

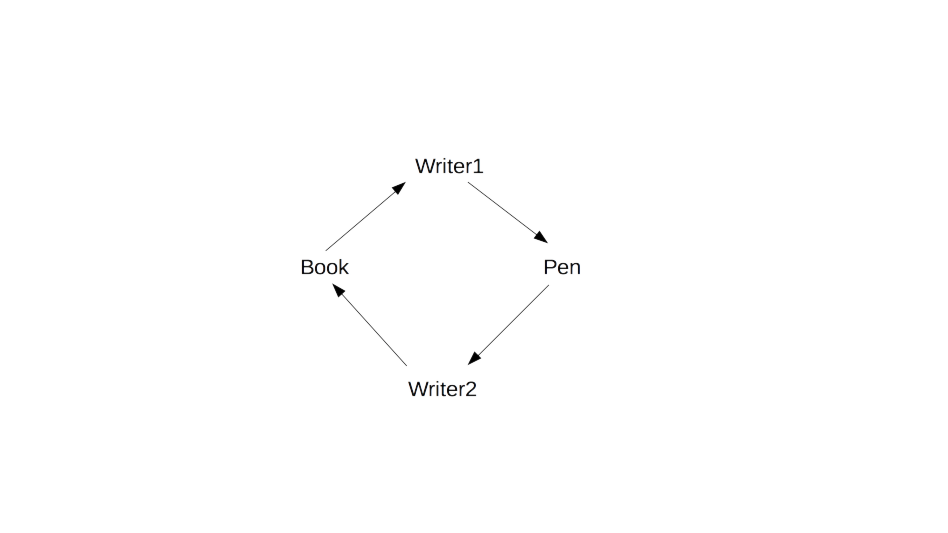
You can see that **list** object is locked by **Thread1** and hence even when the control switched to **Thread2** it can not proceed as the lock is with Thread1. And **Thread1** will add the element where as **Thread2** will fail.

**So when it comes to synchronizing operations synchronized blocks are always better choice over synchronized methods.**

**Deadlocks and solution with lock sequencing**

Deadlock -

Two threads are said to be in a deadlock when both the threads are circularly waiting for a lock over the object and hence they both get into a situation where they can not proceed with the execution.



Assume that for a writer to write lock should be acquired on both the Book and the Pen. And also assume there is only one instance of the book and the pen is available. In that case assume that writer1 acquires lock over the book object and writer 2 acquires lock over the pen object. Now writer1 is waiting for the pen and writer2 is waiting for the book, and they both are deadlocked.

Simulated Example -

1. class Writer1 extends Thread {
3. Object book;
4. Object pen;
6. public Writer1(Object book, Object pen) {
7. this.book = book;
8. this.pen = pen;
9. }
11. @Override
12. public void run() {
14. synchronized(book) {
15. try { Thread.sleep(10); } catch(Exception e) {}
16. synchronized(pen) {
17. System.out.println("Writer1 writing");
18. }
19. }
20. }
21. }
23. class Writer2 extends Thread {
25. Object book;
26. Object pen;
28. public Writer2(Object book, Object pen) {
29. this.book = book;
30. this.pen = pen;
31. }
33. @Override
34. public void run() {
36. synchronized(pen) {
37. try { Thread.sleep(10); } catch(Exception e) {}
38. synchronized(book) {
39. System.out.println("Writer2 writing");
40. }
41. }
42. }
43. }
45. public class Main {
47. public static void main(String[] args) {
48. Object book = new Object();
49. Object pen = new Object();
51. new Writer1(book, pen).start();
52. new Writer2(book, pen).start();
54. System.out.println("Main is done");
55. }
56. }

**Solution -**

Lock sequencing is one possible solution for deadlock avoidance. Adjust the lock sequence for Writer2

1. @Override
2. public void run() {
3. synchronized(book) {
4. try { Thread.sleep(10); } catch(Exception e) {}
5. synchronized(pen) {
6. System.out.println("Writer2 writing");
7. }
8. }
9. }

**Notes - Reentrant Locks**

Read/Write Lock -

Read/Write Lock gives us more flexibility during locking and unlocking. Based on the type of operation being performed over the object we can segregate the locks into

1) readLock

2) writeLock

readLock allows us to lock the object for read operation, and the interesting point is that the read operation can be shared i.e if two threads are waiting for readLock then both of them can proceed forward with the operation as read operation doesn't change the data.

Where as writeLock is mutually exclusive i.e. if a writeLock is accepted then all the other lock requests should wait till the thread that owns the lock releases it.

For example let us assume the following chronologically ordered lock requests

*T1 -> lock.readLock();*

*T2 -> lock.readLock();*

*T3 -> lock.readLock();*

*T4 -> lock.writeLock();*

*T5 -> lock.readLock();*

Here T1, T2, T3 can share the readLock and proceed forward with the operation.  Where T4 should wait till T1, T2 and T3 unlocks.

**Why T5 is waiting ?**

Because writeLock is requested by T4 before its request and hence all subsequent requests to read/write locks should wait.

This is in contrast to synchronized methods/blocks because for synchronized method/block there is no segregation of read and write operations. Object is locked no matter whether it is read or write.

**Caution - It is always better to put the unlock operation in finally, as you need to unlock irrespective of exceptions.**

Example -

Example is just for demo, hence lock/unlock operations are kept in incr() method itself. They can be added to getX() and setX() operations as well.

1. import java.util.concurrent.locks.Lock;
2. import java.util.concurrent.locks.ReadWriteLock;
3. import java.util.concurrent.locks.ReentrantReadWriteLock;
5. class Sample {
7. private int x;
9. // ReadWriteLock object for requesting the lock.
10. ReadWriteLock rw\_lock = new ReentrantReadWriteLock();
12. public int getX() {
13. return x;
14. }
16. public void setX(int x) {
17. this.x = x;
18. }
20. public void incr() {
22. // Request the write lock as the
23. // operation is intended to modify the data.
25. Lock lock = rw\_lock.writeLock();
26. lock.lock();
28. try {
30. int y = getX();
31. y++;
33. // Just for simulation
34. try { Thread.sleep(1); } catch(Exception e) {}
36. setX(y);
38. } finally {
39. // Unlock
40. lock.unlock();
41. }
42. }
44. }
46. class MyThread extends Thread {
48. Sample obj;
50. public MyThread(Sample obj) {
51. this.obj = obj;
52. }
54. public void run() {
55. obj.incr();
56. }
57. }
59. public class Main {
61. public static void main(String[] args) {
63. Sample obj = new Sample();
64. obj.setX(10);
66. MyThread t1 = new MyThread(obj);
67. MyThread t2 = new MyThread(obj);
69. t1.start();
70. t2.start();
72. try {
73. t1.join();
74. t2.join();
75. } catch (InterruptedException e) {
76. e.printStackTrace();
77. }
79. System.out.println( obj.getX() );
80. }
81. }

**Problem Set - 2**

A Map contains Train number and no of seats available. Demonstrate concurrency problem during reserveTickets(n) operation and provide solution.

1. Map<String, Integer> trainInfo = new HashMap<String,Integer>();
2. trainInfo.put("a",100);
3. trainInfo.put("b",100);
5. ....
6. class TicketReservationSystem {
7. public void reserveTicket(String trainName, int ticketCount) {
8. // show problem and define a solution
9. }
10. }

**Problem Set - 2 - Solution**

Note - This is only for demonstration purpose.

1. import java.util.HashMap;
2. import java.util.Map;
4. public class Main {
6. public static void main(String[] args) throws InterruptedException {
8. TicketReservationSystem reservationSystem =
9. new TicketReservationSystem();
10. String trainName = "a";
11. int ticketCount = 50;
13. // These two threads are trying to book 50 tickets for train "a".
14. // After the operation expected available tickets are 0.
15. Thread t1 = new Thread(new TicketBookingTask(
16. reservationSystem,
17. trainName,
18. ticketCount) );

21. Thread t2 = new Thread(new TicketBookingTask(
22. reservationSystem,
23. trainName,
24. ticketCount) );
26. t1.start();
27. t2.start();
29. // Wait for both the threads to complete.
30. t1.join();
31. t2.join();
33. // Expected result is 0, but you might find 50.
34. System.out.println(
35. "Available tickets for train - " + trainName +
36. " are " + reservationSystem.getAvailableTickets(trainName) );
38. }
39. }
41. /\*
42. \* Class that is responsible for reserving/booking tickets.
43. \*/
44. class TicketReservationSystem {
46. Map<String, Integer> reservationInfo = new HashMap<String,Integer>();
48. public TicketReservationSystem() {
49. init();
50. }
52. private void init() {
53. reservationInfo.put("a", 100);
54. reservationInfo.put("b", 100);
55. }
57. /\*
58. \* Reserves the tickets for the given train,
59. \* if there are sufficient tickets available.
60. \*/
61. public void reserveTickets(String trainName, int ticketCount) {
62. Integer available = reservationInfo.get(trainName);
64. if (available == null) {
65. System.out.println("Invalid train name " + trainName);
66. return;
67. }
69. if (available < ticketCount) {
70. System.out.println(
71. "Could not reserve, requested seats not available");
72. return;
73. }
75. // For demo purpose.
76. // Intentionally passed on control to other thread
77. // before updating available tickets.
78. // This is to demonstrate that what if control is switched here..
79. Thread.yield();
81. // Reduce the available ticket count.
82. available -= ticketCount;
84. // Update the Map.
85. reservationInfo.put(trainName, available);
86. }
88. /\*
89. \* Returns available tickets associated with the given trainName.
90. \*/
91. public Integer getAvailableTickets(String trainName) {
92. return reservationInfo.get(trainName);
93. }
94. }

97. /\*
98. \* A Runnable task class that enables parallel ticket booking.
99. \*/
100. class TicketBookingTask implements Runnable {
102. private TicketReservationSystem reservationSystem;
103. private String trainName;
104. private int ticketCount;
106. public TicketBookingTask(TicketReservationSystem reservationSystem,
107. String trainName,
108. int ticketCount) {
109. this.reservationSystem = reservationSystem;
110. this.trainName = trainName;
111. this.ticketCount = ticketCount;
112. }
114. @Override
115. public void run() {
116. reservationSystem.reserveTickets(trainName, ticketCount);
117. }
119. }

If you are finding the result 50, this is because both the threads saw the available tickets are 100 and both of them subtracted 50

i.e. t1 saw available = 100 and

                  available -= 50 =>  50

also t2 saw available = 100 (because before t1 updates the available count, control switched over to other thread. ) and

                 available -= 50 => 50

Both the threads updated the available count as 50. Hence you saw 50.

Solution :-

Though control is passed to other thread because t1 is in the middle of the update operation, we can not allow t2 to proceed with the update i.e. we need to block the threads from simultaneously performing updates over the same object. This can be achieved through synchronized methods/blocks or you can use locks.

Change the above reserveTickets method and make it synchronized

i.e.

1. public synchronized void reserveTickets(String trainName, int ticketCount)

This will block t2 from performing the operation while t1 is in progress or may be the other way around.

You can also use the synchronized block or locks for this as well. Obtain the writeLock before proceeding with update.

**Producer and Consumer Problem**

Thread signalling using wait() and notify() -

Threads can signal each other using the wait and notify methods. wait(), notify() and notifyAll() are the methods of the class Object and hence they are part of all the objects. Just think about a scenario where one thread waits for a signal from some other thread in order to proceed with the execution.

**wait()** method Releases the lock over the object and takes the thread to WAITING state. And the thread remains in that state until some other thread calls the notify() method over the same object. Once notify() is invoked it ends the wait for one single thread and takes the thread to BLOCKED state where the thread remains in that state till the lock is obtained. wait() only returns after obtaining the lock.

**wait(long millis)**  slightly differs, as it takes thread to TIMED\_WAITING and waits only for the specified duration.

**notify()**- notifies one single thread where as **notifyAll()** notifies all the threads waiting for the signal.

*Note - In order to call the wait and notify methods the corresponding thread should hold the lock on the object using synchronized method or block.*

Producer and Consumer Problem -

Here the problem that we are dealing with is the slow consumer, problem occurs when the producer produces the messages at a faster rate than the consumer can consume.

How to solve this problem?

There are different ways of solving it, one approach is to ask the producer to wait till the message is consumed.

Example -

1. // Producer and Consumer problem
2. //    producer --> messageQueue --> consumer.
4. import java.util.ArrayList;
5. import java.util.List;
7. class MessageQueue {
9. List<String> messages;
10. int limit;
12. public MessageQueue(int limit) {
13. messages = new ArrayList<String>();
14. this.limit = limit;
15. }
17. public boolean isFull() {
18. return messages.size() == limit;
19. }
21. public boolean isEmpty() {
22. return messages.size() == 0;
23. }
25. public synchronized void enqueue(String msg) {
27. while (isFull()) {
28. try {
29. this.wait();
30. } catch (InterruptedException e) {
31. // TODO Auto-generated catch block
32. e.printStackTrace();
33. }
34. }
36. messages.add(msg);
37. this.notify();
38. }
40. public synchronized String dequeue() {
42. while (isEmpty()) {
43. try {
44. this.wait();
45. } catch (InterruptedException e) {
46. // TODO Auto-generated catch block
47. e.printStackTrace();
48. }
49. }
51. String message = messages.remove(0);
52. this.notify();
53. return message;
54. }
56. }
58. class ProducerThread extends Thread {
59. MessageQueue queue;
61. public ProducerThread(MessageQueue queue) {
62. this.queue = queue;
63. }
65. @Override
66. public void run() {
67. for(int i=1; i <= 10; i++) {
68. String msg = "Hello-" + i;
69. queue.enqueue(msg);
70. System.out.println("Produced - " + msg);
71. }
72. }
73. }
75. class ConsumerThread extends Thread {
76. MessageQueue queue;
78. public ConsumerThread(MessageQueue queue) {
79. this.queue = queue;
80. }
82. @Override
83. public void run() {
84. for(int i=1; i<=10; i++) {
85. String message = queue.dequeue();
86. System.out.println("Consumed - " + message);
87. }
88. }
89. }
91. public class Main {
92. public static void main(String[] args) throws InterruptedException {
93. MessageQueue queue = new MessageQueue(1);
94. new ProducerThread(queue).start();
95. new ConsumerThread(queue).start();
96. }
97. }

Explanation -

Here we have a class MessageQueue which acts as a buffer between producer thread and the consumer thread. And the buffer is bounded, so we can set the limit during object creation.

**isFull()** returns true if messages size reaches the limit.

**isEmpty()** return true if messages size is equal to zero.

**enqueue()** - Here it is invoked only by ProducerThread; it adds (at the end) the message to the messages array if it is not full, if full it calls the wait() over the queue object (this) till the consumer consumes the message and notifies it. Also once it adds the message to the queue, it calls the notify() in order to end the consumer's wait() if any.

**dequeue()**  - Here it is invoked only by ConsumerThread; it removes the first element and returns it. If the queue is empty it calls the wait() over the queue object till the producer produces the message. Once it consumes the message it calls the notify() to end the producer's wait() if any.

**main()** initiates these operations, by creating the MessageQueue object and passing it to both the producer and consumer threads.

**Notes - ThreadLocal**

ThreadLocal -

***ThreadLocal is considered an anti-pattern please refrain from using it directly.***

ThreadLocal allows us to associate an object with the Thread. It is normally used to hold the information which should be accessible anywhere during the thread execution.

Important Methods -

**get() -**Returns the value associated with the thread.

**set(T value) -**Associates the value with the thread.

**remove() -**Removes the value associated with the thread.

SimulatedExample -

Assume we have two requests, and we assigned two WorkerThreads for servicing them, think that this is a server application assigning a client request to a thread. Now inside the WorkerThread assume that the request is delegated to the framework. And Framework does the pre-processing. During the pre-processing step think that it extracts the user information from the request and sets up the UserContext. The setUserName operation sets the userName in the ThreadLocal object. And then it forwards the request to the API. Assume API calls the services Service1 and Service2, and in between with a delay.Now assume Service1 want to know the user information then it can use the UserContext.getUserName() which retrieves the userName from the thread local object.

And the most important thing is that once the task is done, we need to clear this value. The postProcess step will clear the context i.e. It removes the value associated with the current Thread. May not be important for this example, but in normal applications the same thread is used to service multiple client request, if not cleared the existing value associated with the thread might interfere with some other client request.

1. public class Main {
3. public static void main(String[] args) {
4. String request1 = "a";
5. String request2 = "b";
7. new WorkerThread(request1).start();
8. new WorkerThread(request2).start();
9. }
10. }
12. class UserContext {
13. private static ThreadLocal<String> userInfoThreadLocal =
14. new ThreadLocal<String>() {
15. public String initialValue() {
16. return null;
17. }
18. };
20. public static String getUserName() {
21. return userInfoThreadLocal.get();
22. }
24. public static void setUserName(String userName) {
25. userInfoThreadLocal.set(userName);
26. }
28. public static void clear() {
29. userInfoThreadLocal.remove();
30. }
31. }
33. class WorkerThread extends Thread {
35. String request;
36. Framework framework = new Framework();
38. public WorkerThread(String request) {
39. this.request = request;
40. }
42. @Override
43. public void run() {
44. framework.delegate(request);
45. }
46. }
48. class Framework {
50. API api = new API();
52. public void delegate(String request) {
53. preProcess(request);
54. try {
55. api.process(request);
56. } finally {
57. postProcess();
58. }
59. }
61. private void preProcess(String request) {
62. String userName = getUserName(request);
63. UserContext.setUserName(userName);
64. }
66. private void postProcess() {
67. UserContext.clear();
68. }
70. private String getUserName(String request) {
71. return request;
72. }
73. }
75. class API {
77. Service1 service1 = new Service1();
78. Service2 service2 = new Service2();
80. public void process(String request) {
81. service1.doService1();
82. sleep();
83. service2.doService2();
84. }
86. private void sleep() {
87. try {
88. Thread.sleep(3000);
89. } catch(InterruptedException e) {}
90. }
91. }
93. class Service1 {
95. public void doService1() {
96. System.out.println(
97. "Performing service1 for user - "
98. + UserContext.getUserName()
99. + " " + Thread.currentThread());
100. }
101. }
103. class Service2 {
105. public void doService2() {
106. System.out.println(
107. "Performing service2 for user - "
108. + UserContext.getUserName()
109. + " " + Thread.currentThread());
110. }
111. }

**Output -**

You can see in the output that Thread-0 and Thread-1 has different usernames as each thread has its own copy of the userName.

1. Performing service1 for user - b Thread[Thread-1,5,main]
2. Performing service1 for user - a Thread[Thread-0,5,main]
3. Performing service2 for user - b Thread[Thread-1,5,main]
4. Performing service2 for user - a Thread[Thread-0,5,main]

**Summary -**

ThreadLocal could be used to supply some vital information to all the components involved in processing the request, where this could not be passed directly. But it considered an anti-pattern and creates trouble during asynchronous processing if not handled properly.

**Notes - BlockingQueue and revised producer and consumer problem**

java.util.concurrent package

Has several utilities which help developing multithreaded application.

* ExecutorService
* Callable interface
* Future object
* Locks
* BlockingQueue implementations
* Few other utilities.

BlockingQueue

BlockingQueue is an interface that extends Queue interface. And the implementations include

* ArrayBlockingQueue
  + It is bounded i.e. you need to specify the size.
  + operate on FIFO logic i.e. first in first out, which means that the first inserted element will be the first to be removed.
* LinkedBlockingQueue
  + Optionally Bounded, based on linked nodes i.e. nothing but the linked list.
  + It too operates on FIFO logic.
* PriorityBlockingQueue
  + Unbounded
  + Objects should be Comparable or you should provide a Comparator.

* And there are few other implementations as well.

BlockingQueue operations

Operations that throw Exception if the operation fails.

* add(o)
  + It tries to add an element and if there is no sufficient capacity available this method will throw an exception.
* remove(o)
  + Removes the element that matches with the given object it compares the elements using equals method.
* element()
  + Returns the head element with out removing it. But element() method throws an exception if queue is empty

Operations that return a boolean value with out exception

* offer(o)
  + Returns true if the element is added otherwise false.
* poll()
  + Removes the head element of the queue and returns it, if queue is empty it returns null.
* peek()
  + Returns the head element with out removing it, it returns null if queue is empty.

Operations that block.

* put(o)
  + It will add the element to the queue, but if the queue is full, then it will block the thread till the space is available.
* take()
  + Returns the head element of the queue, if queue is empty this method will block the thread till an element is available.

And the methods with timeout

* offer(o, timeout, timeunit)
* poll(timeout, timeunit)

Revised Producer and Consumer Example -

1. import java.util.concurrent.ArrayBlockingQueue;
2. import java.util.concurrent.BlockingQueue;
4. // Changed from MessageQueue to BlockingQueue.
6. class ProducerThread extends Thread {
7. BlockingQueue<String> queue;
9. public ProducerThread(BlockingQueue<String> queue) {
10. this.queue = queue;
11. }
13. @Override
14. public void run() {
15. for(int i=1; i <= 10; i++) {
16. String msg = "Hello-" + i;
18. // Blocks the thread until the space is available.
19. try {
20. queue.put(msg);
21. System.out.println("Produced - " + msg);
22. } catch (InterruptedException e) {
23. e.printStackTrace();
24. }
26. }
27. }
28. }
30. class ConsumerThread extends Thread {
31. BlockingQueue<String> queue;
33. public ConsumerThread(BlockingQueue<String> queue) {
34. this.queue = queue;
35. }
37. @Override
38. public void run() {
39. for(int i=1; i<=10; i++) {
40. String message = null;
42. // Blocks the thread until the element is available.
43. try {
44. message = queue.take();
45. System.out.println("Consumed - " + message);
46. } catch (InterruptedException e) {
47. e.printStackTrace();
48. }
50. }
51. }
52. }
54. public class Main {
55. public static void main(String[] args) throws InterruptedException {
56. BlockingQueue<String> queue = new ArrayBlockingQueue<String>(1);
57. new ProducerThread(queue).start();
58. new ConsumerThread(queue).start();
59. }
60. }

**Notes - PriorityBlockingQueue**

PriorityBlockingQueue -

PriorityBlockingQueue orders the elements through natural order if they are Comparable or we should use the Comparator.

Example -

1. import java.util.Comparator;
2. import java.util.concurrent.PriorityBlockingQueue;
4. class Student {
5. String name;
6. int rank;
8. public Student(String name, int rank) {
9. this.name = name;
10. this.rank = rank;
11. }
13. public int getRank() {
14. return this.rank;
15. }
17. public String toString() {
18. return String.format("name : %s, rank : %d", name, rank);
19. }
20. }
22. // Compares the Student objects based on the rank field value.
23. class StudentComparator implements Comparator<Student> {
24. @Override
25. public int compare(Student o1, Student o2) {
26. return o1.getRank() - o2.getRank();
27. }
28. }
30. public class Main {
32. public static void main(String[] args) {
33. PriorityBlockingQueue<Integer> queue = new PriorityBlockingQueue<Integer>();
34. queue.add(10);
35. queue.add(2);
36. queue.add(5);
38. System.out.println(queue.poll());
39. System.out.println(queue.poll());
40. System.out.println(queue.poll());
42. PriorityBlockingQueue<Student> queue1 =
43. new PriorityBlockingQueue<Student>(5, new StudentComparator());
45. queue1.add(new Student("a", 12));
46. queue1.add(new Student("b", 1));
47. queue1.add(new Student("c", 4));
49. System.out.println(queue1.poll());
50. System.out.println(queue1.poll());
51. System.out.println(queue1.poll());
52. }
53. }

**Output -**

1. 2
2. 5
3. 10
4. name : b, rank : 1
5. name : c, rank : 4
6. name : a, rank : 12

**Fork Join Framework**

Fork Join Framework -

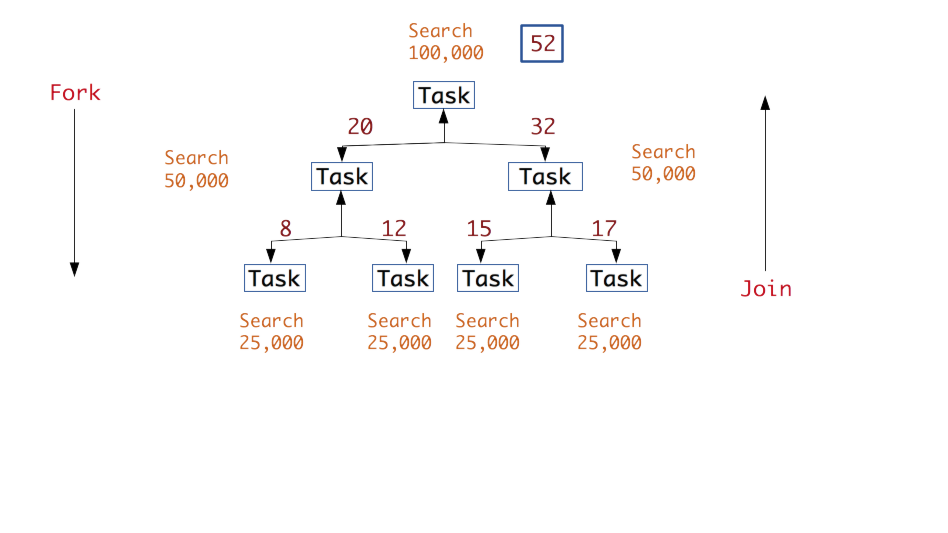
* Available from Java7
* An ExecutorService for ForkJoinTasks
* It differs from ExecutorService by virtue of employing Work-stealing. i.e. if a worker thread has no tasks in the pipeline it will take the task from the task queue of the other busy thread so that the workload is efficiently balanced.
* To access the pool, A static common pool is available for the application and it can be accessed through commonPool() method of the ForkJoinPool class. Using the commonPool is the preferred approach because creating multiple thread pools might have an adverse impact on the performance of the application.

**ForkJoinPool pool = ForkJoinPool.commonPool();**

* For applications which need separate thread pools, we can construct the ForkJoinPool using the level of parallelism needed and by default it is equal to the number of processors.

**ForkJoinPool pool = new ForkJoinPool();**

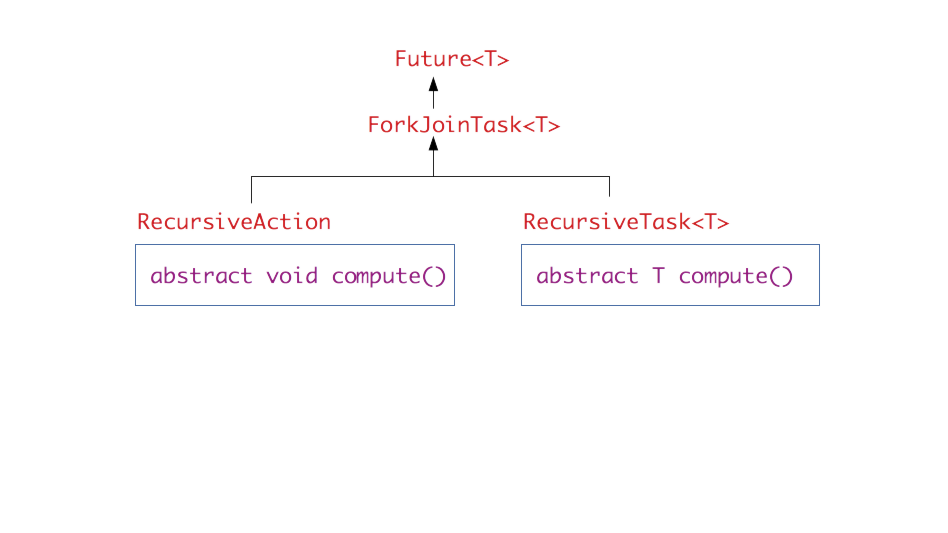
What is fork, join ?



The primary thought process behind fork is that each task is recursively split into subtasks and executed in parallel where as the join operation will wait for the completion of the task and combines the obtained results. To make it simpler, lets assume that the task is to search 100,000 unordered elements and returns the number of occurrences of an element. Assume that, a task will perform the search operation on its own if the list size is 25000. Here the first tasks splits the total elements into two sets of size 50000  and at second level task will split that into two sets of size 25000 each,  and at this level it will not fork any subtasks because the size of the dataset is 25000 and as per our condition we shouLd not create sub tasks.

This is how recursion works as well, there will be a base condition and upon reaching that condition recursive call would be stopped. Similarly here we should put a threshold condition where we will not further fork.

Types of ForkJoinTask(s) -



ForkJoinTask implements Future interface so we can use it to extract the results once the task is done. ForkJoinTask is further divided into two subtypes one is RecursiveAction and RecursiveTask. These classes are abstract; and when you extend you need to override the compute method. If it is a RecursiveAction the compute operation doesn't return any value. Where as in case of RecursiveTask we can return a value. You can relate that with Runnable and Callable.

Submitting the tasks from outside non-fork join clients -

1. YourForkJoinTask task = ......;
2. ForkJoinPool pool = ForkJoinPool.commonPool();
4. // (1) Arranges for asynchronous execution of the given task.
5. pool.execute(task);
7. // (2) Invokes the task, waits for completion and returns the
8. // result.
9. result = pool.invoke(task);
11. // (3) Submits the task for execution
12. pool.submit(task);
14. // Later to obtain the result in case of (1) and (3).
15. Result = task.get();

Submitting the tasks from with in the fork join computations -

1. YourForkJoinSubTask subTask = ......;
3. // (1) Submit the task for execution
4. subTask.fork();
6. // join - waits for the task to complete the computation
7. // and returns its result.
8. result = subTask.join();
10. // (2) Direct approach - invoke the task and wait for result.
11. result = subTask.invoke();

Example -

1. import java.util.concurrent.ForkJoinPool;
2. import java.util.concurrent.RecursiveTask;
4. class SearchTask extends RecursiveTask<Integer> {
6. int arr[];
7. int start, end;
8. int searchEle;
10. public SearchTask(int arr[], int start, int end, int searchEle) {
11. this.arr = arr;
12. this.start = start;
13. this.end = end;
14. this.searchEle = searchEle;
15. }
17. @Override
18. protected Integer compute() {
19. System.out.println(Thread.currentThread());
20. int size = end - start + 1;
21. if (size > 3) {
22. int mid = (end + start) / 2;
23. SearchTask task1 = new SearchTask(arr, start, mid, searchEle);
24. SearchTask task2 = new SearchTask(arr, mid + 1, end, searchEle);
25. task1.fork();
26. task2.fork();
27. int result = task1.join() + task2.join();
28. return result;
29. } else {
30. return processSearch();
31. }
32. }
34. private Integer processSearch() {
35. int count = 0;
36. for(int i = start; i <= end; i++) {
37. if (arr[i] == searchEle) {
38. count++;
39. }
40. }
41. return count;
42. }
43. }
45. public class Main {
47. public static void main(String[] args) {
48. int arr[] = {6, 2, 6, 4, 5, 6, 7, 8, 6, 10, 11, 6};
49. int searchEle = 6;
50. int start = 0;
51. int end = arr.length - 1;
53. ForkJoinPool pool = ForkJoinPool.commonPool();
54. SearchTask task = new SearchTask(arr, start, end, searchEle);
55. int result = pool.invoke(task);
57. System.out.printf("%d found %d times", searchEle, result);
58. }
59. }