### Concurrency

**Concurrency** is an ability to do more than one thing at the same time.

**Multithreading** is multiple threads of same program running concurrently.

**Multithreading is primarily designed to work with single CPU and utilize idle time of the CPU.**

Multithreading reduces idle time of the CPU which improves performance of the application.

**It two or more processors are there; multithreading won’t be able to utilize multiprocessor** but parallel programming using fork-join framework can utilize multiprocessor available in Computer.

**Thread Pool**

Thread Pool is collection of threads, which are created to perform certain tasks.

Thread creation is costly IO operation. It is not advisable to create and destroy threads every now and then.

Instead of creating new thread, when new task arrives, a thread pool keeps number of idle threads that are ready for executing tasks as needed.

After a thread completes execution of a task, it does not die. Instead it remains idle in the pool waiting to be chosen for executing new tasks.

In **parallel execution**, each thread is executed in a separate processing core. Therefore, tasks are really executed in true parallel fashion.

In **concurrent execution**, the threads are executed on a same core. that means tasks are actually executed in interleave fashion, sharing processing time of a processing core.

**Executor - java.util.concurrent.Executor**

[Java executor framework](https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html) (java.util.concurrent.Executor), released with the JDK 5 is **used to run the Runnable objects** **without creating new threads every time** and **mostly re-using the already created threads**.

Executor **creates pool of threads and manages life cycle of all threads** in it.

It **decouples the details of thread creation**, scheduling etc **from the task submission** so you can **focus on developing the task's business logic** **without caring about the thread management details**.

**Executor** object that **executes submitted Runnable task**.

An executor is an object that is responsible for thread management and execution of Runnable tasks submitted from the client code.

**For Example, rather than invoking new Thread (new RunnableTask()).start() for each set of tasks.**

**The java.util.concurrent package defines three executor interfaces:**

**Executor**: is the super type of all executors. It defines only **one method execute (Runnable)**. It’s **replacement for a common thread-creation idiom**.

**ExecutorService**: is an executor that allows tracking the progress of value-returning tasks(callable) via Future object and manage the termination of threads.

**The**[ExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html)**interface provides submit() method. Like execute, submit accepts runnable objects, also accepts callable objects, which allows tasks to return value.**

**The submit() method returns a future object, which is used to retrieve the callable return value and to manage the status of both callable and runnable tasks.**

**ExecutorService also provides methods for submitting large collections of Callable objects. Finally, ExecutorService provides a number of methods for managing the shutdown of the executor**.

Its Key methods include submit () and shutdown ().

**ScheduledExecutorService**: **is an executor service that can schedule tasks to execute after a given delay, or to execute periodically.**

Its key methods are schedule(), scheduleAtFixedRate() and scheduleWithFixedDelay().

**Executors** class is a **factory for executor implementation**.

ExecutorService executorService = Executors.**newSingleThreadExecutor**();

A singleThreadPool uses a single thread to execute tasks.

ExecutorService executorService = Executors.**newCacheThreadPool**();

It will create new threads as they are needed and reuses thread that have become free.

ExecutorService executorService = Executors.**newFixedThreadPool**(2);

It reuses the fixed number of threads.

ExecutorService scheduleExecutorService = Executors.**newScheduledThreadPool**(2);

A scheduledThreadPoolExecutor enables tasks to be executed after a delay or at repeating intervals.

executor.execute(runnable);

executorService.submit(runnable);

executorService.submit(callable);

executorService.shutdown ();

scheduledExecutorService.schedule(task, 5, TimeUnit.SECONDS);

scheduteExecutorService.shutdown ();

**import** java.util.concurrent.Executor;

**public** **class** NewThreadExecutor **implements** Executor {

**public** **void** execute(Runnable command) {

Thread t = **new** Thread(command);

t.start();

}

}

**import** java.util.concurrent.Callable;

**import** java.util.concurrent.Executor;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**public** **class** ExecutorExample1 {

**public** **static** **void** main(String[] args) {

Runnable r = ()->{ System.***out***.println("From Runnable interface it is running"); };

Thread t = **new** Thread(()->{

System.***out***.println("From thread class it is running");

}

);

Thread t2 = **new** Thread(r);

Callable<Void> c =()->{ System.***out***.println("From Callable interface it is running"); **return** **null**; };

Executor e = **new** NewThreadExecutor();

e.execute(r);

e.execute(t);

e.execute(t2);

//e.execute(c); //executor can not run callable interface

ExecutorService executor = Executors.*newFixedThreadPool*(2);

executor.submit(c);

executor.shutdown();

}

}

**Output:**

From Runnable interface it is running

From Runnable interface it is running

From thread class it is running

From Callable interface it is running

**Fork Join framework**

The fork-join framework provides a highly specialized ExecutorService.

The other ExecutorService instances you have seen so far are centred on the concept of submitting multiple tasks to an ExecutorService.

Sometimes you don’t have multiple tasks; instead, you have only big task.

Certain type of larger tasks can be spilt up into smaller subtasks; those subtasks might, in turn, split up into even smaller tasks.

The Fork Join framework **allows to break a certain task** **on several workers** and then **wait for the result to combine them.** It leverages **multiprocessor machine capacity to great extent**.

**Fork**

Fork is a process in which **a task splits into smaller and independent sub tasks which can be executed concurrently**.

**Join**

Join is a process in which **a task join all the results of sub tasks once subtasks have finished executing, otherwise it keeps waiting**.

**Fork Join Pool**

It is a special thread pool designed to work with for-and-join task splitting.

ForkJoinPool fjp = new ForkJoinPool (4);

Here a new ForkJoinPool with parallelism level of 4 CPUs.

**RecursiveAction**

Recursive Action **represents a task which does not return any value.**

Class Writer extends RecursiveAction {

Protected void compute (){}

}

**RecursiveTask**

RecursiveTask **represents a task which does return a value**.

**import** java.util.concurrent.RecursiveTask;

**public** **class** factorial **extends** RecursiveTask<Integer> {

**private** **int** n;

factorial(**int** n) {

**this**.n = n;

}

**public** Integer compute() {

**if** (n <= 1) {

**return** n;

}

factorial t1 = **new** factorial(n - 1);

factorial t2 = **new** factorial(n - 2);

t1.fork();

t2.fork();

**return** t2.join()+ t1.join();

}

**public** **static** **void** main(String args[]) {

factorial f = **new** factorial(8);

System.***out***.println("Output : "+f.compute()); //Output : 21

}

}

**Countdown Latch - java.util.concurrent.CountdownLatch**

- A CountDownLatch **allows one thread to wait for other threads to complete some actions.**

- There might be some situation where we might like our thread to wait one or more threads completes certain operation.

- **A CountDownLatch initialized with an integer count** which represents number of threads to wait for.

- **Count specifies the number of events that must occur before latch is released.**

- **Every time event happens** count is reduced by 1, once the **count reaches zero**, **latch is released**.

- **If the current count in CountDownLatch object is zero**, it immediately will invoke the await () method. otherwise, **the thread blocks until the countdown reaches zero**.

- **Each of thread is performing some actions** will **invoke the countdown () method** to **signal the completion of some actions**.

- **Each instance of CountDownLatch object is good for one-time use**.

**import** java.util.concurrent.CountDownLatch;

**public** **class** CountDownLatchExample {

**public** **static** **void** main(String[] args) **throws** Exception {

CountDownLatch countDownLatch =**new** CountDownLatch(3);

System.***out***.println("CountDownLatch has been created with count=3");

Thread t = **new** Thread(()-> {

**for**(**int** i=2;i>=0;i--){

countDownLatch.countDown();

System.***out***.println(Thread.*currentThread*().getName()+

" has reduced latch count to : "+ i);

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

},"GuruThread");

t.start();

countDownLatch.await();

System.***out***.println("count has reached zero, "+

Thread.*currentThread*().getName()+" thread has ended");

}

}

**OUTPUT:**

CountDownLatch has been created with count=3

GuruThread has reduced latch count to : 2

GuruThread has reduced latch count to : 1

GuruThread has reduced latch count to : 0

count has reached zero, main thread has ended

**Cycle Barrier - java.util.concurrent.CycleBarrier**

Cycle Barrier **allows one or more threads are wait for each other** **to reach common synchronization point.**

- **Each of threads will invoke the await() method which will block** and **wait for the other threads is set to arrive at the barrier point**.

- A CycleBarrier is initialized with an integer count **which represents the fixed number of threads in the set**.

- **when all the threads have reached** (When all the threads have called await() method), **all the waiting threads are released, and event can be triggered as well**.

- **Each instance of cyclic barrier object is cyclic** as they **can be re-used again once all the threads in the set have crossed the barrier point.**

- **CyclicBarrier** is used for **multiple synchronization points** where as **CountDownLatch** is **one synchronization point** **because we can't reuse it.**

**import** java.util.concurrent.BrokenBarrierException;

**import** java.util.concurrent.CyclicBarrier;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**public** **class** CycleBarrierExample3 {

**public** **static** **void** main(String[] args) **throws** Exception{

Thread cycleBarrierEventThread = **new** Thread(()-> { System.***out***.println("cycleBarrierEvent is triggered");});

//CyclicBarrier barrier = new CyclicBarrier(2);

CyclicBarrier barrier = **new** CyclicBarrier(2, cycleBarrierEventThread);

ExecutorService exe = Executors.*newFixedThreadPool*(2);

Runnable run =()->{

**try** {

System.***out***.println("Runnable thread");

barrier.await();

System.***out***.println("After Barrier Point reached... it runs");

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (BrokenBarrierException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

};

exe.submit(run);

exe.submit(run);

exe.submit(run);

exe.submit(run);

System.***out***.println("Main thread");

exe.shutdown();

}}

**output**

Main thread

Runnable thread

Runnable thread

cycleBarrierEvent is triggered

After Barrier Point reached... it runs

After Barrier Point reached... it runs

Runnable thread

Runnable thread

cycleBarrierEvent is triggered

After Barrier Point reached... it runs

After Barrier Point reached... it runs

**Phaser**

Phaser is a **reusable synchronization barrier**, it is somewhat similar in functionality of CycleBarrier and CountDownLatch, but it provides more flexibility than both of them.

The Phaser is **class that allows you to control the execution of tasks divided into phases**. **None of the tasks advance to the next phase until all of the tasks have finished the current phase**.

**The basic task of Phaser is that it registers the task, start and then deregisters**.

**In CycleBarrier, we used to register parties in the constructo**r, but **phaser provides us flexibility of registering and deregistering the parties at any time**.

**A phaser allows one or more threads to wait for the other threads in the set to arrive at common synchronization point called a phase**.

**To participate in the synchronization**, a **Thread has to invoke the register() method on an instance of phaser**.

**A Thread will invoke the arriveAndAwaitAdvance() method to block and wait for other threads in the set to arrive at the barrier point.**

**arriveAndDeregister () in Java Phaser**

Current thread (Party) Arrives and deregisters from phaser. **Deregistration reduces the number of parties that may be required in future to move to next phase**.

**Like the CycleBarrier, an instance of phaser object can be reused again once all the threads in the set have crossed the barrier point.**

- Dynamic number of parties

- Reusable

- Advanceable : phase.arrive() that how it is advance.

**Lock - java.util.concurrent.lock.Lock**

Lock provides **more flexible synchronization operation** **than synchronized method**.

**The responsibility of acquiring as well as releasing the lock is on the programmer**.

**There are different kinds of locks:**

1. **Re-entrant lock** - to implement a lock that can be associated with condition.

2. **ReentrantReadWriteLock** - which separates read and write operation.

3. **StampedLock** - new features of Java 8 that includes three modes for controlling read/write access.

The use of Lock () method to acquire a lock behaves the same as the use of synchronized keyword.

you release the lock by calling the unlock() method of the Lock interface after you are done with lock.

**tryLock**() method tries to acquire the lock without pausing the thread, that is, If the thread could not acquire the lock because it was held by some other thread, then it returns immediately instead of waiting for the lock to be released.

You can also specify the timeout in trylock () method to wait for the lock to be available.

lock.tryLock (1,TimeUnit.SECONDS);

The thread will now pause for 1 second and wait for the lock to be available. If the lock couldn't be acquired within 1 second, then thread returns.

**A Thread acquires the lock multiple times recursively, thus the word 'ReEntrant'.**

**ReentrantReadWriteLock**

ReentrantReadWriteLock class that implements ReadWriteLock interface.

This class **has two locks, one for read operation and one for write operation**.

There **can be more than one threads using read operations simultaneously**, but **only one thread can be using write operations**, **there can't be any thread doing read operations**.

A **ReadWriteLock** implementation guarantees the following behaviours.

- Multiple threads can read the data at the same time, as long as there are no threads is updating the data.

- Only one thread can update the data at a time, causing other threads (both read and writes) block until the write block is released.

- If a thread attempts to update the data, while other threads are reading, the write thread also blocks until the read lock is released.

**ReadWriteLock** can be used to add concurrency feature to data structures, but it doesn't guarantee the performance.

ReadWriteLock lock = new ReadWriteReentrantLock();

lock.readLock().lock(); lock.writeLock().lock();

lock.readLock().unLock(); lock.writeLock().unLock();

**StampedLock** It differentiates between exclusive and non-exclusive locks, similar to the ReentrantReadWriteLock. However, it also allows for optimistic reads, which is not supported by the ReentrantReadWriteLock.

Most of the time read operations doesn't need to wait for write operation completion and as a result of this, the full-fledged read lock isn't required.

public String readWithOptimisticLock(String key) {

    long stamp = lock.tryOptimisticRead();

    String value = map.get(key);

    if(!lock.validate(stamp)) {

        stamp = lock.readLock();

        try {

            return map.get(key);

        } finally {

            lock.unlock(stamp);

        }

    }

    return value;

}

 Instead of the usual locking, it returns a long number whenever a lock is granted. This stamp number is then used to unlock again.

StampedLock sl = new StampedLock();

long stamp = sl.writeLock();

try {

// do something that needs exclusive access

} finally {

sl.unlockWrite(stamp);

}

**There are few differences between the lock and synchronized block that are given below.**

1. Lock interface provides the guarantee of sequence in which the waiting thread will be given access, whereas the synchronized block doesn’t guarantee it.

2. Lock interface provides the option of timeout if the lock is not granted whereas the synchronized block does not provide that.

3. **The methods of lock interface**, (Lock() and unlock()) methods **can be called different methods** whereas **single synchronized block must be fully contained in a single method**.

The use of synchronized keyword requires that a **thread should acquire and release an object's monitor lock in the same block of code.**

When you use synchronized keyword to acquire an object's monitor lock, the lock is released by the JVM when the program leaves the block in which the lock was acquired.

**import** java.util.concurrent.locks.Lock;

**import** java.util.concurrent.locks.ReentrantLock;

**public** **class** ReentranctLockExample {

**public** **static** **void** main(String[] args) {

Lock lock = **new** ReentrantLock();

Thread runnableThread = **new** Thread(() ->{

System.***out***.println(Thread.*currentThread*().getName()

+" is Waiting to acquire lock");

lock.lock();

System.***out***.println(Thread.*currentThread*().getName()

+" has acquired lock.");

**try** {

Thread.*sleep*(5000);

System.***out***.println(Thread.*currentThread*().getName()

+" is sleeping.");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println(Thread.*currentThread*().getName()

+" has released lock.");

lock.unlock();

});

**new** Thread(runnableThread,"Thread1").start();

**new** Thread(runnableThread,"Thread2").start();

}

}

**Output:**

Thread1 is Waiting to acquire lock

Thread1 has acquired lock.

Thread2 is Waiting to acquire lock

Thread1 is sleeping.

Thread1 has released lock.

Thread2 has acquired lock.

Thread2 is sleeping.

Thread2 has released lock.

|  |  |  |
| --- | --- | --- |
| Executor.execute(runnable) | ExecutorService.submit(runnable);  ExecutorService.submit(callable);  executorService.shutdown(); | scheduleExecutorService. schedule()  scheduleExecutorService. scheduleAtFixedRate()  scheduleExecutorService. scheduleWithFixedDelay(task,5,TimeUnit.seconds). |

|  |  |  |
| --- | --- | --- |
| Lock lock = new Lock();  Lock.lock();  Lock.tryLock();  Lock.unlock(); | ReentrantReadWriteLock rrwl = new ReentrantReadWriteLock();  rrwl.readLock().lock();  rrwl.readLock().unLock();  rrwl.writeLock().lock();  rrwl.writeLock().unLock(); | StampedLock sl = new StampedLock();  Long stamp = sl.readLock();  Sl.readUnLock(stamp);  Long stamp = sl.writeLock();  Sl.unLockWrite(stamp);  Long stamp = sl.tryOptimisticRead(); |

|  |  |
| --- | --- |
| **Semaphore** | Acquire()  Release() |
| **Atomic Variable –**  AtomicInteger  AtomicLong  AtomicBoolean | incrementAndGet()  decrementAndGet()  getAndUpdate()  updateAndGet() |
| **CountDownLatch** | Await()  countdown() |
| **CycleBarrier** | Await() |
| **Phaser** | Register()  arriveAndAwaitAdvance();  ArriveAndDeregister() |
| **Executors** | ExecutorService es = Executors.newSingleThreadPool();  ExecutorService es = Executors.newCachedThreadPool(2);  ExecutorService es = Executors.newFixedThreadPool(2);  ExecutorService es = Executors.newFixedThreadPool(2); |
| **ForkJoinPool** | ForkJoinPool fjp = new ForkJoinPool(4);  instances.fork();  instances.join();  **RecursiveAction**  Public void compute();  **RecursiveTask**  Public int compute(); |

**Semaphore**

Semaphore is used **to control the number of threads that are using resources**. That resource can be something like a shared data, or any file.

**The Count on a semaphore can go up and down** as different threads call **acquire**() and **release**().

A semaphore maintains a counter to **specify the number of resources** **that the semaphore controls**.

**Access to the resource is allowed if the counter is greater than zero**, while **zero value of the counter indicates that no resource is available at the moment and so the access is denied**.

Semaphore s = new Semaphore(2);

s.acquire(); // Acquires a permit from semaphore if available; otherwise, it blocks until a permit becomes available.

s.release(); // Release a permit from semaphore basically decrement the value.

**Synchronized allows only one thread of execution to access the resource at the same time.**

**Semaphore allows up to n threads of execution to access the resource at the same time.**

**import** java.util.concurrent.Semaphore;

**public** **class** SemaphoreProducerConsumer {

**static** Semaphore *sProducer* = **new** Semaphore(1);

**static** Semaphore *sConsumer* = **new** Semaphore(0);

**public** **static** **void** main(String[] args) {

Thread producer = **new** Thread(()-> {

**for**(**int** i=1;i<=5;i++){

**try** {

*sProducer*.acquire();

System.***out***.println("Produced : "+i);

*sConsumer*.release();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

},"ProducerThread");

Thread consumer = **new** Thread(()->{

**for**(**int** i=1;i<=5;i++) {

**try** {

*sConsumer*.acquire();

System.***out***.println("Consumed : "+i);

*sProducer*.release();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

});

producer.start();

consumer.start();

}

}

**Atomic Variable**

Atomic Variable is used in multithreaded environment when you need to make sure that only one thread can update an variable.

Atomic Variable don't use locks or other synchronization mechanisms to protect the access to their values.

Atomic classes internally use **compare-and-swap instructions** supported **by modern CPUs to achieve synchronization. These instructions are generally much faster than locks.**

**CompareAndSwap (CAS) basically works like optimistic locking**.

Before performing any operation like i++, it first compares the value of the variable with the value actually stored, and

if it is same then it swaps the new value in Atomic way.

If they are not same then it retries the read and get the new value stored and repeat the operation, it make sure that this check and act is done in atomic way.

**Look at the java.util.concurrent.atomic package, you will see the following classes**:

AtomicBoolean

AtomicInteger

AtomicLong

**They are called atomic variables because they provide some operations that cannot be interfered by multiple threads.** Here's to name a few:

**incrementAndGet()** : atomically increment by one the current value.

**decrementAndGet**() : atomically decrement by one the current value.

These operations **are guaranteed to execute atomically using machine-level instruction** on modern processors.

AtomicInteger ai = new AtomicInteger(0);

int n = ai.incrementAndGet();

AtomicInteger has been extended in Java 8 with the following two methods

int getAndUpdate()

int updateAndGet()

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.atomic.AtomicInteger;

**public** **class** AtomicIntegerExample {

**public** **static** **void** main(String[] args) {

AtomicInteger counter = **new** AtomicInteger(50);

Runnable runnable =()->{

System.***out***.println("Value for Thread " + Thread.*currentThread*().getName() +

" Before increment " + counter.get());

counter.incrementAndGet();

System.***out***.println("Value for Thread " + Thread.*currentThread*().getName() +

" After increment " + counter.get());

};

// Starting 6 threads

ExecutorService ex = Executors.*newFixedThreadPool*(6);

ex.execute(runnable);

ex.execute(runnable);

ex.execute(runnable);

ex.execute(runnable);

ex.execute(runnable);

ex.execute(runnable);

//shutting down the executor service

ex.shutdown();

}

}

**output**

Value for Thread pool-1-thread-2 Before increment 50

Value for Thread pool-1-thread-2 After increment 51

Value for Thread pool-1-thread-5 Before increment 50

Value for Thread pool-1-thread-6 Before increment 50

Value for Thread pool-1-thread-6 After increment 53

Value for Thread pool-1-thread-4 Before increment 50

Value for Thread pool-1-thread-4 After increment 54

Value for Thread pool-1-thread-3 Before increment 50

Value for Thread pool-1-thread-1 Before increment 50

Value for Thread pool-1-thread-1 After increment 56

Value for Thread pool-1-thread-3 After increment 55

Value for Thread pool-1-thread-5 After increment 52

**volatile**

- **Volatile can be used only with variables. It doesn't acquire any lock on variable. It will never create any deadlock in program.**

- **If a variable or field is declared volatile in that case Java memory model ensures that all threads see consistent value for the variable.**

- **All the threads will read its value from memory and will not cache it.**

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**public** **class** VolatileExample {

**public** **static** **void** main(String[] args) {

Data data = **new** Data();

//public volatile int counter = 0; //illegal modifier for the counter parameter

Runnable runnable =()->{

System.***out***.println("Value for Thread " + Thread.*currentThread*().getName() +

" Before increment " + data.getCounter());

data.incrementCounter();

System.***out***.println("Value for Thread " + Thread.*currentThread*().getName() +

" After increment " + data.getCounter());

};

// Starting 3 threads

ExecutorService ex = Executors.*newFixedThreadPool*(6);

ex.execute(runnable);

ex.execute(runnable);

ex.execute(runnable);

//shutting down the executor service

ex.shutdown();

}

}

**class** Data{

**public** **volatile** **int** counter = 0;

**public** **int** getCounter() {

**return** counter;

}

**public** **void** incrementCounter() {

++counter;

}

}

**Output:**

Value for Thread pool-1-thread-1 Before increment 0

Value for Thread pool-1-thread-1 After increment 1

Value for Thread pool-1-thread-3 Before increment 0

Value for Thread pool-1-thread-3 After increment 2

Value for Thread pool-1-thread-2 Before increment 0

Value for Thread pool-1-thread-2 After increment 3

**ThreadLocal**

**ThreadLocal** is used to create thread-local variables.

ThreadLocal enables you to create variables that can only be read and written by the same thread. Even if two threads set different values on the same thread local object, they can't see each others values.

ThreadLocal<Integer> threadLocal = new ThreadLocal<>();

threadLocal.set((int)Math.random());

System.out.println("output :"+threadLocal.get());

int a= threadLocal.get();

**Future**

The submit () method of the ExecutorService submits the task for execution by thread. However, it does not know when the result of submitted task will be available. Therefore, it returns a special type of value called a Future which **can be used to fetch the result of the task when it is available**.

Future<String> future = executorService.submit (callable);

String result = future.get ();

**You will notice the delay in output because Future.get() method waits for the Java Callable task to complete**.

You can cancel a Future using future.cancel() method. It attempts to cancel the execution of the task and return true if it is cancelled successfully, otherwise, it returns false.

List<Callable<String>) taskList = Arrays.asList(task1,task2,task3);

List<Future<String>) futures = executorService.invokeAll(taskList);

for(Future<String> future:futures){

System.out.println(future.get());

}

executorService.shutdown();

**BlockingQueue - java.util.concurrent.BlockingQueue**

BlockingQueue is a queue that supports operation that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element.

**import** java.util.concurrent.BlockingQueue;

**import** java.util.concurrent.LinkedBlockingQueue;

**public** **class** ProducerConsumerBlockingQueue {

**public** **static** **void** main(String[] args) {

BlockingQueue<Integer> sharedQueue = **new** LinkedBlockingQueue<Integer>(5);

Thread producer = **new** Thread(()-> {

System.***out***.println("Produced Test : ");

**for**(**int** i=1;i<=3;i++){

**try** {

System.***out***.println("Produced : "+i);

sharedQueue.put(i);

**if**(i==1)

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

},"ProducerThread");

Thread consumer = **new** Thread(()->{

**while**(**true**) {

**try** {

System.***out***.println("Consumed test");

System.***out***.println("Consumed : "+sharedQueue.take());

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

});

consumer.start();

producer.start();

//consumer.start();

}

}

**OUTPUT:**

Consumed test

Produced Test :

Produced : 1

Consumed : 1

Consumed test

Produced : 2

Produced : 3

Consumed : 2

Consumed test

Consumed : 3

Consumed test