Concurrency

Main Components/Utilities of Concurrent Package

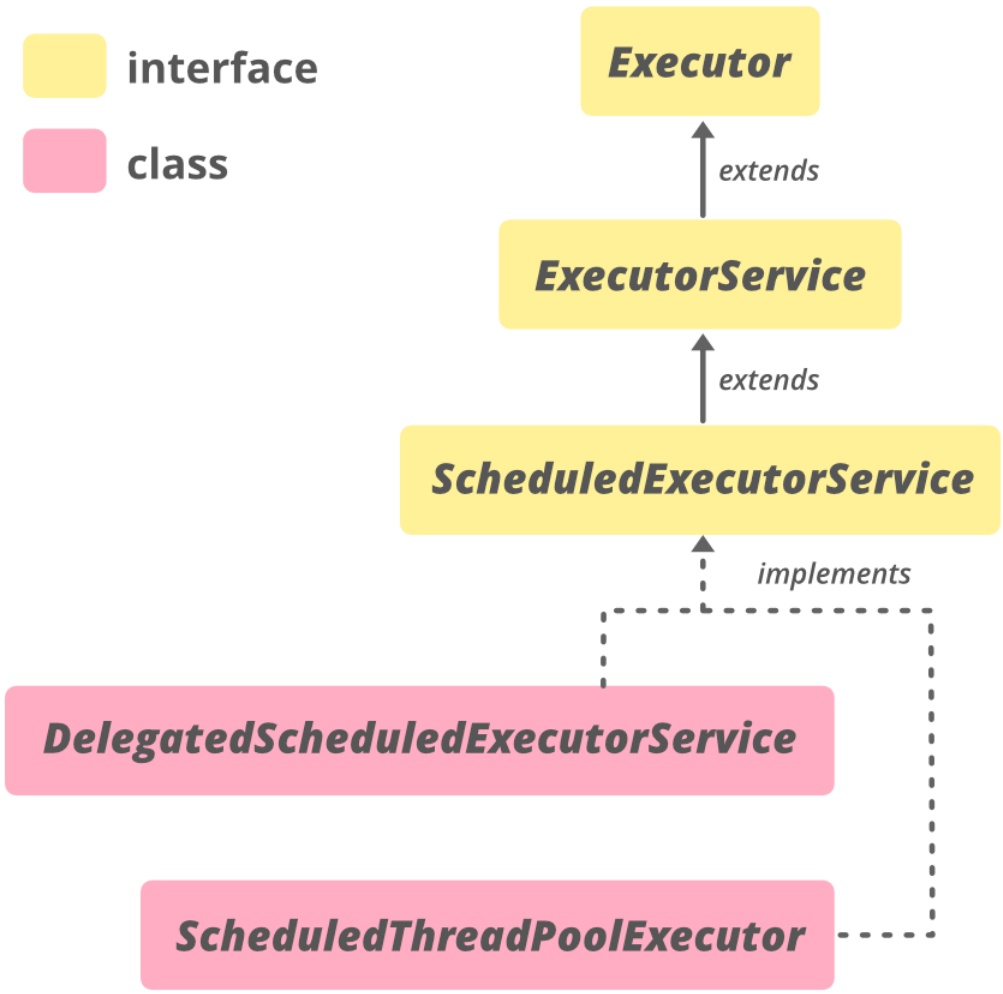
▼ What is Excecutor Framework?

Java executor framework (***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.

The ***java.util.concurrent.Executors*** provide factory methods that are being used to create **ThreadPools** of worker threads. Thread pools overcome this issue by keeping the threads alive and reusing the threads. Any excess tasks flowing in that the threads in the pool can handle are held in a Queue. Once any of the threads get free, they pick up the next task from this queue. This task queue is essentially unbounded for the out-of-box executors provided by the JDK.

Some types of Java Executors are listed:

1. SingleThreadExecutor
2. FixedThreadPool(n)+
3. CachedThreadPool
4. ScheduledExecutor



▼ What is ExecutorService?

The **ExecutorService i**nterface extends Executor by adding methods that help manage and control the execution of threads. It is defined in ***java.util.concurrent package***. It defines methods that execute the threads that return results, a set of threads that determine the shutdown status. 、

The **ExecutorService interface** is implemented in a utility class called **Executors**. It defines methods that provide an implementation of the ExecutorService interface and many other interfaces, with some default settings.

▼ What is ScheduledExecutorService?

The **ScheduledExecutorService** interface in Java is a sub-interface of **ExecutorService** interface defined in **java.util.concurrent** package. This interface is used to run the given tasks periodically or once after a given delay. The ScheduledExecutorService interface has declared some useful methods to schedule the given tasks. These methods are implemented by the **ScheduledThreadPoolExecutor** class.

> https://www.geeksforgeeks.org/scheduledexecutorservice-interface-in-java/

▼ What is future

Future interface provides methods **to check if the computation is complete, to wait for its completion and to retrieve the results of the computation**. The result is retrieved using Future's get() method when the computation has completed, and it blocks until it is completed. Future and FutureTask both are available in **java.util.concurrent package**from Java 1.5

▼ CountDownLatch

CountDownLatch is used to make sure that a task waits for other threads before it starts.

When we create an object of CountDownLatch, we specify the number of threads it should wait for, all such thread are required to do count down by calling CountDownLatch.countDown() once they are completed or ready to the job. As soon as count reaches zero, the waiting task starts running.

▼ CyclicBarrier

CyclicBarrier is used to make threads wait for each other. It is used when different threads process a part of computation and when all threads have completed the execution, the result needs to be combined in the parent thread. In other words, a CyclicBarrier is used when multiple thread carry out different sub tasks and the output of these sub tasks need to be combined to form the final output. After completing its execution, threads call await() method and wait for other threads to reach the barrier. Once all the threads have reached, the barriers then give the way for threads to proceed.

> https://www.geeksforgeeks.org/java-util-concurrent-cyclicbarrier-java/

▼ Semaphore

A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

Working of semaphore

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit.

- If the semaphore's count is greater than zero, then the thread acquires a permit, which causes the semaphore's count to be decremented.
- Otherwise, the thread will be blocked until a permit can be acquired.
- When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore's count to be incremented.
- If there is another thread waiting for a permit, then that thread will acquire a permit at that time.

Java provide **Semaphore** class in *java.util.concurrent* package that implements this mechanism, so you don't have to implement your own semaphores.

> https://www.geeksforgeeks.org/semaphore-in-java/

▼ ThreadFactory

The **ThreadFactory** interface defined in the **java.util.concurrent** package is based on the **factory design pattern**. As its name suggests, it is used to create new threads on demand. Threads can be created in two ways:

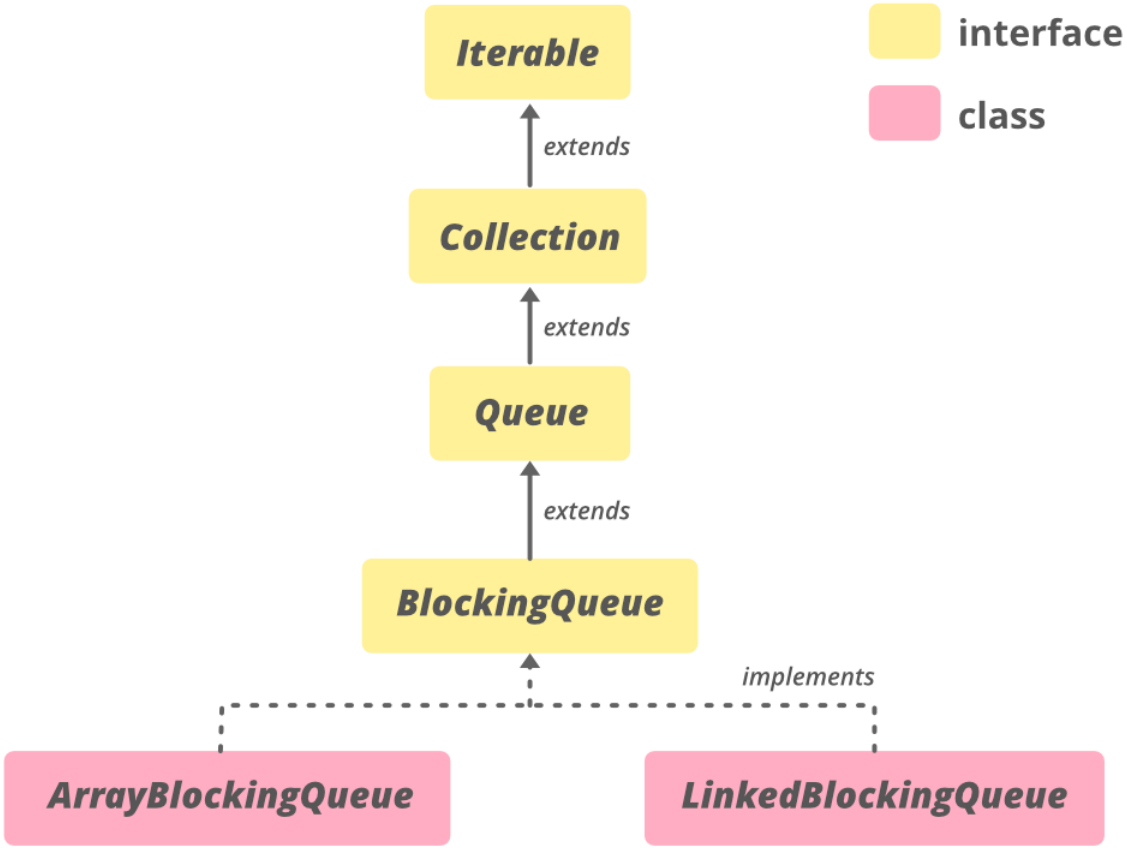
1. Creating a class that extends the Thread class and then creating its objects.

2. Creating a class that implements the Runnable interface and then using its object to create threads.

> https://www.geeksforgeeks.org/threadfactory-interface-in-java-with-examples/

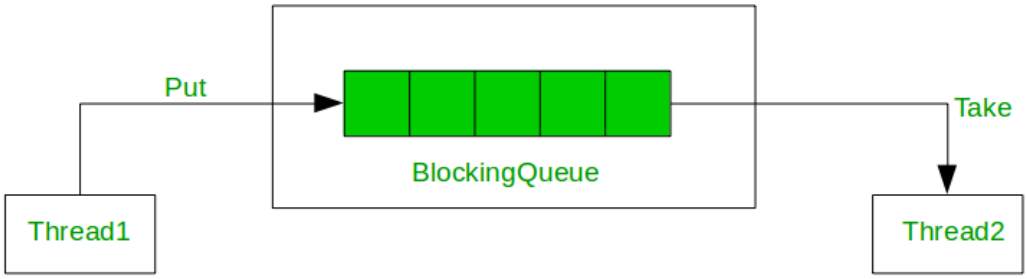
▼ BlockingQueue

The **BlockingQueue interface** in Java is added in Java 1.5 along with various other concurrent Utility classes like **ConcurrentHashMap**, **Counting Semaphore**, **CopyOnWriteArrrayLis**t, etc.



BlockingQueue interface supports flow control (in addition to queue) by introducing blocking if either BlockingQueue is full or empty. A thread trying to enqueue an element in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more elements or clearing the queue completely. Similarly, it blocks a thread trying to delete from an empty queue until some other threads insert an item. BlockingQueue does not accept a null value. If we try to enqueue the null item, then it throws **NullPointerException**. Java provides several

BlockingQueue implementations such as **LinkedBlockingQueue**, **ArrayBlockingQueue**, **PriorityBlockingQueue**, **SynchronousQueue**, etc. Java BlockingQueue interface implementations are thread-safe. All methods of BlockingQueue are atomic in nature and use internal locks or other forms of concurrency control. Java 5 comes with BlockingQueue implementations in the **java.util.concurrent package.**



> https://www.geeksforgeeks.org/blockingqueue-interface-in-java/

.

▼ DelayQueue

The **DelayQueue** class is a member of the **Java Collections Framework.** It belongs to package. DelayQueue implements the **BlockingQueue** interface. 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. DelayQueue head contains the element that has expired in the least time. If no delay has expired, then there is no head and the poll will return null. DelayQueue accepts only those elements that belong to a class of type **Delayed** or those implement **java.util.concurrent.Delayed** interface. The DelayQueue blocks the elements internally until a certain delay has expired. DelayQueue implements the getDelay(TimeUnit.NANOSECONDS) method to return the remaining delay time. The TimeUnit instance passed to the getDelay() method is an Enum that tells which time unit the delay should be returned in. The TimeUnit enum can take DAYS, HOURS, MINUTES, SECONDS, MILLISECONDS, MICROSECONDS, NANOSECONDS. This queue does not permit null elements. This class and its

iterator implement all of the optional methods of the **Collection** and **Iterator** interfaces. The Iterator provided in method **iterator()** is not guaranteed to traverse the elements of the DelayQueue in any particular order.

▼ Lock

A java.util.concurrent.locks.Lock interface is used to as a thread synchronization mechanism similar to synchronized blocks. New Locking mechanism is more flexible and provides more options than a synchronized block. Main differences between a Lock and a synchronized block are following −

- **Guarantee of sequence** − Synchronized block does not provide any guarantee of sequence in which waiting thread will be given access. Lock interface handles it.
- **No timeout** − Synchronized block has no option of timeout if lock is not granted. Lock interface provides such option.
- **Single method** − Synchronized block must be fully contained within a single method whereas a lock interface's methods lock() and unlock() can be called in different methods.

▼ Phaser

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