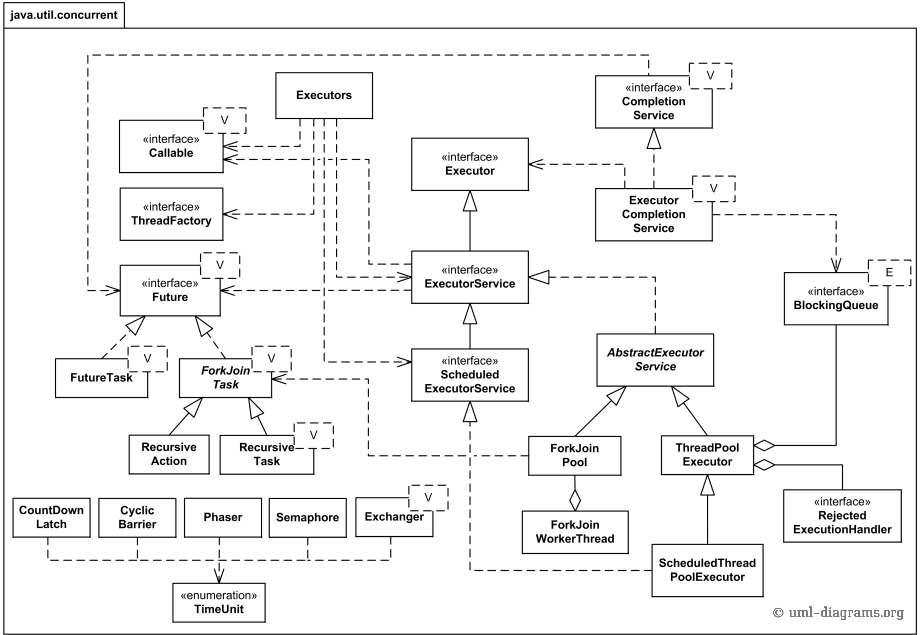
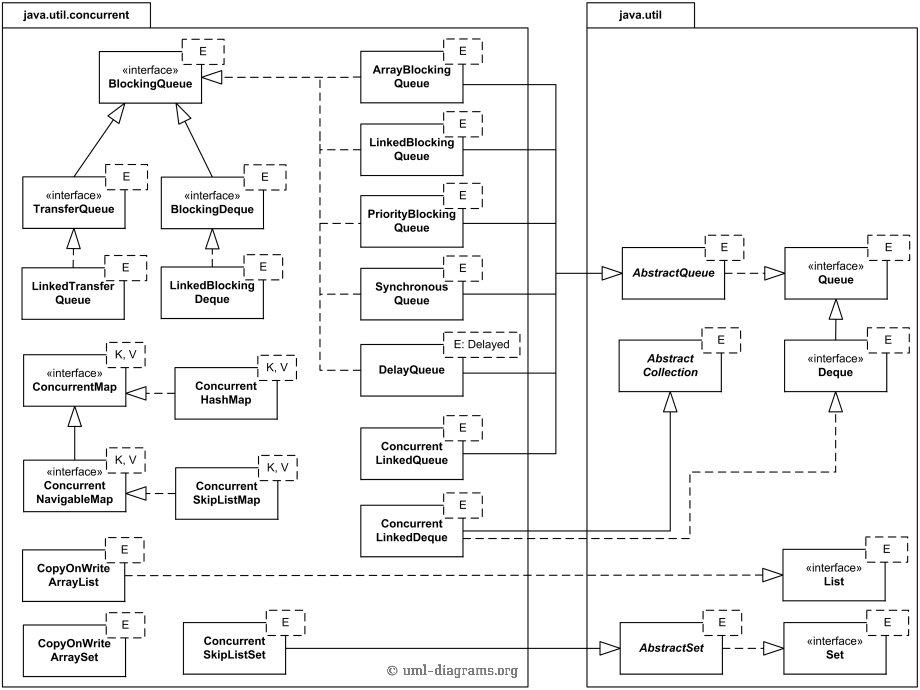
# Java™ 7 util.concurrent API

Here we provide several UML [**class diagrams**](https://www.uml-diagrams.org/class-diagrams-overview.html) **for the Java™ 7 java.util.concurrent package**. Several java.util.concurrent.\* packages introduced with version 5.0 of the Java platform added high-level concurrency features to the Java and new concurrent data structures to the Java Collections Framework.

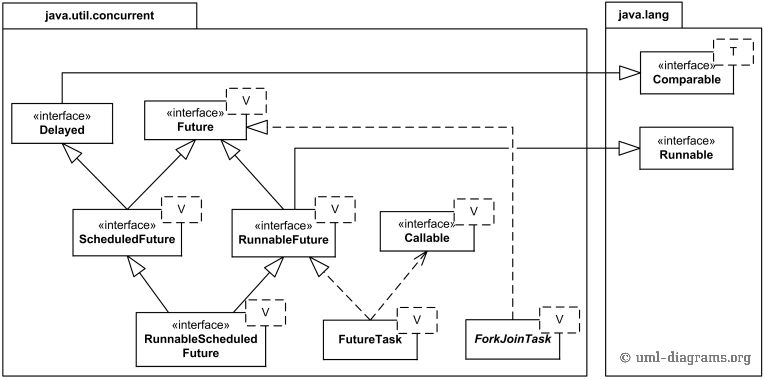


*UML class diagram for the Java™ 7 executors and thread pool managers from the java.util.concurrent package.*

**Executors** define a high-level API for launching and managing threads to support large-scale applications mostly by adding **thread pool management**abilities. The java.util.concurrent package includes several thread pool management implementation classes.



**Concurrent collections** are also part of the java.util.concurrent package. These collections reduce the need for synchronization and designed to support concurrent access and modifications of the large collections of data.



The **Future<V>** interface represents the result of an asynchronous computation, where type V is the result type returned by the Future's get method. Methods of this interface allow to wait for the computation to complete, to cancel execution of the task, to check if the computation is complete or was cancelled, and to retrieve the result of the computation.

The **Delayed** interface allows to mark objects that should be acted upon after a given delay. ScheduledFuture<V> interface extends both Future<V> and Delayed, and is usually a result of scheduling a task with a ScheduledExecutorService.

The **FutureTask** class is an implementation of Future that implements java.lang.**Runnable** as required by RunnableFeature interface, and thus may be executed by an Executor.

# Hierarchy For Package java.util.concurrent

**Package Hierarchies:**

* [All Packages](https://docs.oracle.com/javase/8/docs/api/overview-tree.html)

## Class Hierarchy

* java.lang.[**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)
  + java.util.[**AbstractCollection**](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractCollection.html)**<E> (implements java.util.**[Collection](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)<E>)
    - java.util.[**AbstractQueue**](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html)**<E> (implements java.util.**[Queue](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html)<E>)
      * java.util.concurrent.[**ArrayBlockingQueue**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html)**<E> (implements java.util.concurrent.**[BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)<E>, java.io.[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
      * java.util.concurrent.[**ConcurrentLinkedQueue**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html)**<E> (implements java.util.**[Queue](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html)<E>, java.io.[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
      * java.util.concurrent.[**DelayQueue**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/DelayQueue.html)**<E> (implements java.util.concurrent.**[BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)<E>)
      * java.util.concurrent.[**LinkedBlockingDeque**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingDeque.html)**<E> (implements java.util.concurrent.**[BlockingDeque](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html)<E>, java.io.[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
      * java.util.concurrent.[**LinkedBlockingQueue**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html)**<E> (implements java.util.concurrent.**[BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)<E>, java.io.[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
      * java.util.concurrent.[**LinkedTransferQueue**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedTransferQueue.html)**<E> (implements java.io.**[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html), java.util.concurrent.[TransferQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html)<E>)
      * java.util.concurrent.[**PriorityBlockingQueue**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/PriorityBlockingQueue.html)**<E> (implements java.util.concurrent.**[BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)<E>, java.io.[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
      * java.util.concurrent.[**SynchronousQueue**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/SynchronousQueue.html)**<E> (implements java.util.concurrent.**[BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)<E>, java.io.[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
    - java.util.[**AbstractSet**](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractSet.html)**<E> (implements java.util.**[Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html)<E>)
      * java.util.concurrent.[**ConcurrentSkipListSet**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListSet.html)**<E> (implements java.lang.**[Cloneable](https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html), java.util.[NavigableSet](https://docs.oracle.com/javase/8/docs/api/java/util/NavigableSet.html)<E>, java.io.[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
      * java.util.concurrent.[**CopyOnWriteArraySet**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArraySet.html)**<E> (implements java.io.**[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
    - java.util.concurrent.[**ConcurrentLinkedDeque**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedDeque.html)**<E> (implements java.util.**[Deque](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html)<E>, java.io.[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
  + java.util.concurrent.[**AbstractExecutorService**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/AbstractExecutorService.html) **(implements java.util.concurrent.**[ExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html))
    - java.util.concurrent.[**ForkJoinPool**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html)
    - java.util.concurrent.[**ThreadPoolExecutor**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html)
      * java.util.concurrent.[**ScheduledThreadPoolExecutor**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html) **(implements java.util.concurrent.**[ScheduledExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledExecutorService.html))
  + java.util.[**AbstractMap**](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)**<K,V> (implements java.util.**[Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<K,V>)
    - java.util.concurrent.[**ConcurrentHashMap**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html)**<K,V> (implements java.util.concurrent.**[ConcurrentMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html)<K,V>, java.io.[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
    - java.util.concurrent.[**ConcurrentSkipListMap**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html)**<K,V> (implements java.lang.**[Cloneable](https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html), java.util.concurrent.[ConcurrentNavigableMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentNavigableMap.html)<K,V>, java.io.[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
  + java.util.concurrent.[**CompletableFuture**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html)**<T> (implements java.util.concurrent.**[CompletionStage](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html)<T>, java.util.concurrent.[Future](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html)<V>)
  + java.util.concurrent.[**ConcurrentHashMap.KeySetView**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.KeySetView.html)**<K,V> (implements java.io.**[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html), java.util.[Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html)<E>)
  + java.util.concurrent.[**CopyOnWriteArrayList**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArrayList.html)**<E> (implements java.lang.**[Cloneable](https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html), java.util.[List](https://docs.oracle.com/javase/8/docs/api/java/util/List.html)<E>, java.util.[RandomAccess](https://docs.oracle.com/javase/8/docs/api/java/util/RandomAccess.html), java.io.[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
  + java.util.concurrent.[**CountDownLatch**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountDownLatch.html)
  + java.util.concurrent.[**CyclicBarrier**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html)
  + java.util.concurrent.[**Exchanger**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Exchanger.html)**<V>**
  + java.util.concurrent.[**ExecutorCompletionService**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorCompletionService.html)**<V> (implements java.util.concurrent.**[CompletionService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionService.html)<V>)
  + java.util.concurrent.[**Executors**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html)
  + java.util.concurrent.[**ForkJoinTask**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html)**<V> (implements java.util.concurrent.**[Future](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html)<V>, java.io.[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
    - java.util.concurrent.[**CountedCompleter**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountedCompleter.html)**<T>**
    - java.util.concurrent.[**RecursiveAction**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RecursiveAction.html)
    - java.util.concurrent.[**RecursiveTask**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RecursiveTask.html)**<V>**
  + java.util.concurrent.[**FutureTask**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/FutureTask.html)**<V> (implements java.util.concurrent.**[RunnableFuture](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RunnableFuture.html)<V>)
  + java.util.concurrent.[**Phaser**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Phaser.html)
  + java.util.[**Random**](https://docs.oracle.com/javase/8/docs/api/java/util/Random.html) **(implements java.io.**[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
    - java.util.concurrent.[**ThreadLocalRandom**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadLocalRandom.html)
  + java.util.concurrent.[**Semaphore**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html) **(implements java.io.**[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
  + java.lang.[**Thread**](https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html) **(implements java.lang.**[Runnable](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html))
    - java.util.concurrent.[**ForkJoinWorkerThread**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinWorkerThread.html)
  + java.util.concurrent.[**ThreadPoolExecutor.AbortPolicy**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.AbortPolicy.html) **(implements java.util.concurrent.**[RejectedExecutionHandler](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RejectedExecutionHandler.html))
  + java.util.concurrent.[**ThreadPoolExecutor.CallerRunsPolicy**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.CallerRunsPolicy.html) **(implements java.util.concurrent.**[RejectedExecutionHandler](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RejectedExecutionHandler.html))
  + java.util.concurrent.[**ThreadPoolExecutor.DiscardOldestPolicy**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.DiscardOldestPolicy.html) **(implements java.util.concurrent.**[RejectedExecutionHandler](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RejectedExecutionHandler.html))
  + java.util.concurrent.[**ThreadPoolExecutor.DiscardPolicy**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.DiscardPolicy.html) **(implements java.util.concurrent.**[RejectedExecutionHandler](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RejectedExecutionHandler.html))
  + java.lang.[**Throwable**](https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html) **(implements java.io.**[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
    - java.lang.[**Exception**](https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html)
      * java.util.concurrent.[**BrokenBarrierException**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BrokenBarrierException.html)
      * java.util.concurrent.[**ExecutionException**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutionException.html)
      * java.lang.[**RuntimeException**](https://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html)
        + java.util.concurrent.[**CompletionException**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionException.html)
        + java.lang.[**IllegalStateException**](https://docs.oracle.com/javase/8/docs/api/java/lang/IllegalStateException.html)

java.util.concurrent.[**CancellationException**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CancellationException.html)

* + - * + java.util.concurrent.[**RejectedExecutionException**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RejectedExecutionException.html)
      * java.util.concurrent.[**TimeoutException**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TimeoutException.html)

## Interface Hierarchy

* java.util.concurrent.[**Callable**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html)**<V>**
* java.lang.[**Comparable**](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html)**<T>**
  + java.util.concurrent.[**Delayed**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Delayed.html)
    - java.util.concurrent.[**ScheduledFuture**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledFuture.html)**<V> (also extends java.util.concurrent.**[Future](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html)<V>)
      * java.util.concurrent.[**RunnableScheduledFuture**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RunnableScheduledFuture.html)**<V> (also extends java.util.concurrent.**[RunnableFuture](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RunnableFuture.html)<V>)
* java.util.concurrent.[**CompletableFuture.AsynchronousCompletionTask**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.AsynchronousCompletionTask.html)
* java.util.concurrent.[**CompletionService**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionService.html)**<V>**
* java.util.concurrent.[**CompletionStage**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html)**<T>**
* java.util.concurrent.[**Executor**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html)
  + java.util.concurrent.[**ExecutorService**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html)
    - java.util.concurrent.[**ScheduledExecutorService**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledExecutorService.html)
* java.util.concurrent.[**ForkJoinPool.ForkJoinWorkerThreadFactory**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.ForkJoinWorkerThreadFactory.html)
* java.util.concurrent.[**ForkJoinPool.ManagedBlocker**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.ManagedBlocker.html)
* java.util.concurrent.[**Future**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html)**<V>**
  + java.util.concurrent.[**RunnableFuture**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RunnableFuture.html)**<V> (also extends java.lang.**[Runnable](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html))
    - java.util.concurrent.[**RunnableScheduledFuture**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RunnableScheduledFuture.html)**<V> (also extends java.util.concurrent.**[ScheduledFuture](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledFuture.html)<V>)
  + java.util.concurrent.[**ScheduledFuture**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledFuture.html)**<V> (also extends java.util.concurrent.**[Delayed](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Delayed.html))
    - java.util.concurrent.[**RunnableScheduledFuture**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RunnableScheduledFuture.html)**<V> (also extends java.util.concurrent.**[RunnableFuture](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RunnableFuture.html)<V>)
* java.lang.[**Iterable**](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html)**<T>**
  + java.util.[**Collection**](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)**<E>**
    - java.util.[**Queue**](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html)**<E>**
      * java.util.concurrent.[**BlockingQueue**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)**<E>**
        + java.util.concurrent.[**BlockingDeque**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html)**<E> (also extends java.util.**[Deque](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html)<E>)
        + java.util.concurrent.[**TransferQueue**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html)**<E>**
      * java.util.[**Deque**](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html)**<E>**
        + java.util.concurrent.[**BlockingDeque**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html)**<E> (also extends java.util.concurrent.**[BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)<E>)
* java.util.[**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)**<K,V>**
  + java.util.concurrent.[**ConcurrentMap**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html)**<K,V>**
    - java.util.concurrent.[**ConcurrentNavigableMap**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentNavigableMap.html)**<K,V> (also extends java.util.**[NavigableMap](https://docs.oracle.com/javase/8/docs/api/java/util/NavigableMap.html)<K,V>)
  + java.util.[**SortedMap**](https://docs.oracle.com/javase/8/docs/api/java/util/SortedMap.html)**<K,V>**
    - java.util.[**NavigableMap**](https://docs.oracle.com/javase/8/docs/api/java/util/NavigableMap.html)**<K,V>**
      * java.util.concurrent.[**ConcurrentNavigableMap**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentNavigableMap.html)**<K,V> (also extends java.util.concurrent.**[ConcurrentMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html)<K,V>)
* java.util.concurrent.[**RejectedExecutionHandler**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RejectedExecutionHandler.html)
* java.lang.[**Runnable**](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html)
  + java.util.concurrent.[**RunnableFuture**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RunnableFuture.html)**<V> (also extends java.util.concurrent.**[Future](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html)<V>)
    - java.util.concurrent.[**RunnableScheduledFuture**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RunnableScheduledFuture.html)**<V> (also extends java.util.concurrent.**[ScheduledFuture](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledFuture.html)<V>)
* java.util.concurrent.[**ThreadFactory**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadFactory.html)

## Enum Hierarchy

* java.lang.[**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)
  + java.lang.[**Enum**](https://docs.oracle.com/javase/8/docs/api/java/lang/Enum.html)**<E> (implements java.lang.**[Comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html)<T>, java.io.[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html))
    - java.util.concurrent.[**TimeUnit**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TimeUnit.html)

## Overview

The *java.util.concurrent* package provides tools for creating concurrent applications.

In this article, we will do an overview of the whole package.

## 2. Main Components

The *java.util.concurrent* contains way too many features to discuss in a single write-up. In this article, we will mainly focus on some of the most useful utilities from this package like:

* *Executor*
* *ExecutorService*
* *ScheduledExecutorService*
* *Future*
* *CountDownLatch*
* *CyclicBarrier*
* *Semaphore*
* *ThreadFactory*
* *BlockingQueue*
* *DelayQueue*
* *Locks*
* *Phaser*

You can also find many dedicated articles to individual classes here.

### 2.1. *Executor*

[***Executor***](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html) ***is an interface that represents an object that executes provided tasks.***

It depends on the particular implementation (from where the invocation is initiated) if the task should be run on a new or current thread. Hence, using this interface, we can decouple the task execution flow from the actual task execution mechanism.

One point to note here is that *Executor* does not strictly require the task execution to be asynchronous. In the simplest case, an executor can invoke the submitted task instantly in the invoking thread.

We need to create an invoker to create the executor instance:

|  |
| --- |
| **public** **class** Invoker **implements** Executor {  @Override  **public** **void** execute(Runnable r) {  r.run();  }  } |

Now, we can use this invoker to execute the task.

|  |
| --- |
| **public** **void** execute() {  Executor executor = **new** Invoker();  executor.execute( () -> {  // task to be performed  });  } |

Point to note here is that if the executor can’t accept the task for execution, it will throw [*RejectedExecutionException*](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RejectedExecutionException.html)*.*

### 2.2. *ExecutorService*

*ExecutorService* is a complete solution for asynchronous processing. It manages an in-memory queue and schedules submitted tasks based on thread availability.

To use *ExecutorService,* we need to create one *Runnable* class.

|  |
| --- |
| **public** **class** Task **implements** Runnable {  @Override  **public** **void** run() {  // task details  }  } |

Now we can create the *ExecutorService* instance and assign this task. At the time of creation, we need to specify the thread-pool size.

|  |
| --- |
| ExecutorService executor = Executors.newFixedThreadPool(10); |

If we want to create a single-threaded *ExecutorService* instance, we can use ***newSingleThreadExecutor(ThreadFactory threadFactory)*** to create the instance.

Once the executor is created, we can use it to submit the task.

|  |
| --- |
| **public** **void** execute() {  executor.submit(**new** Task());  } |

We can also create the *Runnable* instance while submitting the task.

|  |
| --- |
| executor.submit(() -> {  **new** Task();  }); |

It also comes with two out-of-the-box execution termination methods. The first one is *shutdown()*; it waits till the all submitted task finish executing. The other method is *shutdownNow()* whic*h* immediately terminates all the pending/executing tasks.

There is also another method *awaitTermination(long timeout, TimeUnit unit)* which forcefully blocks until all tasks have completed execution after a shutdown event triggered or execution-timeout occurred, or the execution thread itself is interrupted,

|  |
| --- |
| **try** {  executor.awaitTermination( 20l, TimeUnit.NANOSECONDS );  } **catch** (InterruptedException e) {  e.printStackTrace();  } |

### 2.3. *ScheduledExecutorService*

*ScheduledExecutorService* is a similar interface to *ExecutorService,* but it can perform tasks periodically.

***Executor and ExecutorService*‘s methods are scheduled on the spot without introducing any artificial delay.** Zero or any negative value signifies that the request needs to be executed instantly.

We can use both *Runnable* and *Callable* interface to define the task.

|  |
| --- |
| **public** **void** execute() {  ScheduledExecutorService executorService  = Executors.newSingleThreadScheduledExecutor();    Future<String> future = executorService.schedule(() -> {  // ...  **return** **"Hello world"**;  }, 1, TimeUnit.SECONDS);    ScheduledFuture<?> scheduledFuture = executorService.schedule(() -> {  // ...  }, 1, TimeUnit.SECONDS);    executorService.shutdown();  } |

*ScheduledExecutorService* can also schedule the task **after some given fixed delay**:

|  |
| --- |
| executorService.scheduleAtFixedRate(() -> {  // ...  }, 1, 10, TimeUnit.SECONDS);    executorService.scheduleWithFixedDelay(() -> {  // ...  }, 1, 10, TimeUnit.SECONDS); |

Here, the ***scheduleAtFixedRate( Runnable command, long initialDelay, long period, TimeUnit unit )*** method creates and executes a periodic action that is invoked firstly after the provided initial delay, and subsequently with the given period until the service instance shutdowns.

The ***scheduleWithFixedDelay( Runnable command, long initialDelay, long delay, TimeUnit unit )*** method creates and executes a periodic action that is invoked firstly after the provided initial delay, and repeatedly with the given delay between the termination of the executing one and the invocation of the next one.

### 2.4. *Future*

***Future* is used to represent the result of an asynchronous operation.** It comes with methods for checking if the asynchronous operation is completed or not, getting the computed result, etc.

What’s more, the *cancel(boolean mayInterruptIfRunning)* API cancels the operation and releases the executing thread. If the value of *mayInterruptIfRunning* is true, the thread executing the task will be terminated instantly.

Otherwise, in-progress tasks will be allowed to complete.

We can use below code snippet to create a future instance:

|  |
| --- |
| **public** **void** invoke() {  ExecutorService executorService = Executors.newFixedThreadPool(10);    Future<String> future = executorService.submit(() -> {  // ...  Thread.sleep(10000l);  **return** **"Hello world"**;  });  } |

We can use following code snippet to check if the future result is ready and fetch the data if the computation is done:

|  |
| --- |
| **if** (future.isDone() && !future.isCancelled()) {  **try** {  str = future.get();  } **catch** (InterruptedException | ExecutionException e) {  e.printStackTrace();  }  } |

We can also specify a timeout for a given operation. If the task takes more than this time, a *TimeoutException* is thrown:

|  |
| --- |
| **try** {  future.get(10, TimeUnit.SECONDS);  } **catch** (InterruptedException | ExecutionException | TimeoutException e) {  e.printStackTrace();  } |

### 2.5. *CountDownLatch*

*CountDownLatch* (introduced in *JDK 5*) is a utility class which blocks a set of threads until some operation completes.

A *CountDownLatch* is initialized with a *counter(Integer* type); this counter decrements as the dependent threads complete execution. But once the counter reaches zero, other threads get released.

You can learn more about *CountDownLatch* [here](https://www.baeldung.com/java-countdown-latch).

### 2.6. *CyclicBarrier*

*CyclicBarrier* works almost same as *CountDownLatch* except that we can reuse it. Unlike *CountDownLatch*, it allows multiple threads to wait for each other using *await()* method(known as barrier condition) before invoking the final task.

We need to create a *Runnable* task instance initiate the barrier condition:

|  |
| --- |
| **public** **class** Task **implements** Runnable {    **private** CyclicBarrier barrier;    **public** Task(CyclicBarrier barrier) {  **this**.barrier = barrier;  }    @Override  **public** **void** run() {  **try** {  LOG.info(Thread.currentThread().getName() +  **" is waiting"**);  barrier.await();  LOG.info(Thread.currentThread().getName() +  **" is released"**);  } **catch** (InterruptedException | BrokenBarrierException e) {  e.printStackTrace();  }  }    } |

Now we can invoke some threads to race for the barrier condition:

|  |
| --- |
| **public** **void** start() {    CyclicBarrier cyclicBarrier = **new** CyclicBarrier(3, () -> {  // ...  LOG.info(**"All previous tasks are completed"**);  });    Thread t1 = **new** Thread(**new** Task(cyclicBarrier), **"T1"**);  Thread t2 = **new** Thread(**new** Task(cyclicBarrier), **"T2"**);  Thread t3 = **new** Thread(**new** Task(cyclicBarrier), **"T3"**);    **if** (!cyclicBarrier.isBroken()) {  t1.start();  t2.start();  t3.start();  }  } |

Here, the *isBroken()* method checks if any of the threads got interrupted during the execution time. We should always perform this check before performing the actual process.

### 2.7. *Semaphore*

The *Semaphore* is used for blocking thread level access to some part of the physical or logical resource. A semaphore contains a set of permits; whenever a thread tries to enter the critical section, it needs to check the semaphore if a permit is available or not.

**If a permit is not available (via *tryAcquire()*), the thread is not allowed to jump into the critical section; however, if the permit is available the access is granted, and the permit counter decreases.**

Once the executing thread releases the critical section, again the permit counter increases (done by *release()* method).

We can specify a timeout for acquiring access by using the *tryAcquire(long timeout, TimeUnit unit)* method.

**We can also check the number of available permits or the number of threads waiting to acquire the semaphore.**

Following code snippet can be used to use implement a semaphore:

|  |
| --- |
| **static** Semaphore semaphore = **new** Semaphore(10);    **public** **void** execute() **throws** InterruptedException {    LOG.info(**"Available permit : "** + semaphore.availablePermits());  LOG.info(**"Number of threads waiting to acquire: "** +  semaphore.getQueueLength());    **if** (semaphore.tryAcquire()) {  semaphore.acquire();  // ...  semaphore.release();  }    } |

We can implement a *Mutex* like data-structure using *Semaphore*. More details on this [can be found here.](https://www.baeldung.com/java-semaphore)

### 2.8. *ThreadFactory*

As the name suggests, *ThreadFactory* acts as a thread (non-existing) pool which creates a new thread on demand. It eliminates the need of a lot of boilerplate coding for implementing efficient thread creation mechanisms.

We can define a *ThreadFactory*:

|  |
| --- |
| **public** **class** BaeldungThreadFactory **implements** ThreadFactory {  **private** **int** threadId;  **private** String name;    **public** BaeldungThreadFactory(String name) {  threadId = 1;  **this**.name = name;  }    @Override  **public** Thread newThread(Runnable r) {  Thread t = **new** Thread(r, name + **"-Thread\_"** + threadId);  LOG.info(**"created new thread with id : "** + threadId +  **" and name : "** + t.getName());  threadId++;  **return** t;  }  } |

We can use this *newThread(Runnable r)* method to create a new thread at runtime:

|  |
| --- |
| BaeldungThreadFactory factory = **new** BaeldungThreadFactory(  **"BaeldungThreadFactory"**);  **for** (**int** i = 0; i < 10; i++) {  Thread t = factory.newThread(**new** Task());  t.start();  } |

### 2.9. *BlockingQueue*

In asynchronous programming, one of the most common integration patterns is the [producer-consumer pattern](https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem). The *java.util.concurrent* package comes with a data-structure know as *BlockingQueue* – which can be very useful in these async scenarios.

More information and a working example on this is available [here](https://www.baeldung.com/java-blocking-queue).

### 2.10. *DelayQueue*

*DelayQueue* is an infinite-size blocking queue of elements where an element can only be pulled if it’s expiration time (known as user defined delay) is completed. Hence, the topmost element (*head*) will have the most amount delay and it will be polled last.

More information and a working example on this is available [here](https://www.baeldung.com/java-delay-queue).

### 2.11. *Locks*

Not surprisingly, *Lock* is a utility for blocking other threads from accessing a certain segment of code, apart from the thread that’s executing it currently.

The main difference between a Lock and a Synchronized block is that synchronized block is fully contained in a method; however, we can have Lock API’s lock() and unlock() operation in separate methods.

More information and a working example on this is available [here](https://www.baeldung.com/java-concurrent-locks).

### 2.12. *Phaser*

*Phaser* is a more flexible solution than *CyclicBarrier* and *CountDownLatch* – used to act as a reusable barrier on which the dynamic number of threads need to wait before continuing execution. We can coordinate multiple phases of execution, reusing a *Phaser* instance for each program phase.

More information and a working example on this is available [here](https://www.baeldung.com/java-phaser).

## 3. Conclusion

In this high-level, overview article, we’ve focused on the different utilities available of *java.util.concurrent* package.

As always, the full source code is available [over on GitHub](https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-concurrency-basic).

### What is atomic operation? What are atomic classes in Java Concurrency API?

Atomic operations are performed in a single unit of task without interference from other operations. Atomic operations are necessity in multi-threaded environment to avoid data inconsistency.

int++ is not an atomic operation. So by the time one threads read it’s value and increment it by one, other thread has read the older value leading to wrong result.

To solve this issue, we will have to make sure that increment operation on count is atomic, we can do that using Synchronization but Java 5 java.util.concurrent.atomic provides wrapper classes for int and long that can be used to achieve this atomically without usage of Synchronization. Go to this article to learn more about [atomic concurrent classes](https://www.journaldev.com/1095/atomicinteger-java).

### What is Lock interface in Java Concurrency API? What are it’s benefits over synchronization?

Lock interface provide more extensive locking operations than can be obtained using synchronized methods and statements. They allow more flexible structuring, may have quite different properties, and may support multiple associated Condition objects.  
The advantages of a lock are

* + it’s possible to make them fair
  + it’s possible to make a thread responsive to interruption while waiting on a Lock object.
  + it’s possible to try to acquire the lock, but return immediately or after a timeout if the lock can’t be acquired
  + it’s possible to acquire and release locks in different scopes, and in different orders
* Read more at [**Java Lock Example**](https://www.journaldev.com/2377/java-lock-example-reentrantlock)**.**

### What is Executors Framework?

In Java 5, Executor framework was introduced with the java.util.concurrent.Executor interface.

The Executor framework is a framework for standardizing invocation, scheduling, execution, and control of asynchronous tasks according to a set of execution policies.

Creating a lot many threads with no bounds to the maximum threshold can cause application to run out of heap memory. So, creating a ThreadPool is a better solution as a finite number of threads can be pooled and reused. Executors framework facilitate process of creating Thread pools in java. Check out this post to learn with example code to [create thread pool using Executors framework](https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice).

### What is BlockingQueue? How can we implement Producer-Consumer problem using Blocking Queue?

java.util.concurrent.BlockingQueue is a Queue that supports operations 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.

BlockingQueue doesn’t accept null values and throw NullPointerException if you try to store null value in the queue.

BlockingQueue implementations are thread-safe. All queuing methods are atomic in nature and use internal locks or other forms of concurrency control.

BlockingQueue interface is part of [java collections framework](https://www.journaldev.com/1260/collections-in-java-tutorial) and it’s primarily used for implementing producer consumer problem.  
Check this post for [producer-consumer problem implementation using BlockingQueue](https://www.journaldev.com/1034/java-blockingqueue-example).

### What is Callable and Future?

Java 5 introduced java.util.concurrent.Callable interface in concurrency package that is similar to Runnable interface but it can return any Object and able to throw Exception.

Callable interface use Generic to define the return type of Object. Executors class provide useful methods to execute Callable in a thread pool. Since callable tasks run in parallel, we have to wait for the returned Object. Callable tasks return java.util.concurrent.Future object. Using Future we can find out the status of the Callable task and get the returned Object. It provides get() method that can wait for the Callable to finish and then return the result.  
Check this post for [Callable Future Example](https://www.journaldev.com/1090/java-callable-future-example).

### What is FutureTask Class?

FutureTask is the base implementation class of Future interface and we can use it with Executors for asynchronous processing. Most of the time we don’t need to use FutureTask class but it comes real handy if we want to override some of the methods of Future interface and want to keep most of the base implementation. We can just extend this class and override the methods according to our requirements. Check out [**Java FutureTask Example**](https://www.journaldev.com/1650/java-futuretask-example-program) **post to learn how to use it and what are different methods it has.**

### What are Concurrent Collection Classes?

Java Collection classes are fail-fast which means that if the Collection will be changed while some thread is traversing over it using iterator, the iterator.next() will throw ConcurrentModificationException.

Concurrent Collection classes support full concurrency of retrievals and adjustable expected concurrency for updates.  
Major classes are ConcurrentHashMap, CopyOnWriteArrayList and CopyOnWriteArraySet, check this post to learn [how to avoid ConcurrentModificationException when using iterator](https://www.journaldev.com/378/java-util-concurrentmodificationexception).

### What is Executors Class?

Executors class provide utility methods for Executor, ExecutorService, ScheduledExecutorService, ThreadFactory, and Callable classes.

Executors class can be used to easily create Thread Pool in java, also this is the only class supporting execution of Callable implementations.

### What are some of the improvements in Concurrency API in Java 8?

Some important concurrent API enhancements are:

* + ConcurrentHashMap compute(), forEach(), forEachEntry(), forEachKey(), forEachValue(), merge(), reduce() and search() methods.
  + CompletableFuture that may be explicitly completed (setting its value and status).
  + Executors newWorkStealingPool() method to create a work-stealing thread pool using all available processors as its target parallelism level.

### What is the Thread’s interrupt flag? How can you set and check it? How does it relate to the InterruptedException?

The interrupt flag, or interrupt status, is an internal *Thread* flag that is set when the thread is interrupted. To set it, simply call *thread.interrupt()* on the thread object*.*

If a thread is currently inside one of the methods that throw *InterruptedException* (*wait*, *join*, *sleep* etc.), then this method immediately throws InterruptedException. The thread is free to process this exception according to its own logic.

If a thread is not inside such method and *thread.interrupt()* is called, nothing special happens. It is thread’s responsibility to periodically check the interrupt status using *static Thread.interrupted()* or instance *isInterrupted()* method. The difference between these methods is that the *static Thread.interrupt()* clears the interrupt flag, while *isInterrupted()* does not.

### Q7. What are Executor and ExecutorService? What are the differences between these interfaces?

*Executor* and *ExecutorService* are two related interfaces of *java.util.concurrent* framework. *Executor* is a very simple interface with a single *execute* method accepting *Runnable* instances for execution. In most cases, this is the interface that your task-executing code should depend on.

*ExecutorService* extends the *Executor* interface with multiple methods for handling and checking the lifecycle of a concurrent task execution service (termination of tasks in case of shutdown) and methods for more complex asynchronous task handling including *Futures*.

For more info on using *Executor* and *ExecutorService*, see the article [A Guide to Java ExecutorService](https://www.baeldung.com/java-executor-service-tutorial).

### Q8. What are the available implementations of ExecutorService in the standard library?

The *ExecutorService* interface has three standard implementations:

* ***ThreadPoolExecutor*** — for executing tasks using a pool of threads. Once a thread is finished executing the task, it goes back into the pool. If all threads in the pool are busy, then the task has to wait for its turn.
* ***ScheduledThreadPoolExecutor*** allows to schedule task execution instead of running it immediately when a thread is available. It can also schedule tasks with fixed rate or fixed delay.
* ***ForkJoinPool*** is a special *ExecutorService* for dealing with recursive algorithms tasks. If you use a regular *ThreadPoolExecutor* for a recursive algorithm, you will quickly find all your threads are busy waiting for the lower levels of recursion to finish. The *ForkJoinPool*implements the so-called work-stealing algorithm that allows it to use available threads more efficiently.

### Q9. What is Java Memory Model (JMM)? Describe its purpose and basic ideas.

Java Memory Model is a part of Java language specification described in [Chapter 17.4](https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4). It specifies how multiple threads access common memory in a concurrent Java application, and how data changes by one thread are made visible to other threads. While being quite short and concise, JMM may be hard to grasp without strong mathematical background.

The need for memory model arises from the fact that the way your Java code is accessing data is not how it actually happens on the lower levels. Memory writes and reads may be reordered or optimized by the Java compiler, JIT compiler, and even CPU, as long as the observable result of these reads and writes is the same.

This can lead to counter-intuitive results when your application is scaled to multiple threads because most of these optimizations take into account a single thread of execution (the cross-thread optimizers are still extremely hard to implement). Another huge problem is that the memory in modern systems is multilayered: multiple cores of a processor may keep some non-flushed data in their caches or read/write buffers, which also affects the state of the memory observed from other cores.

To make things worse, the existence of different memory access architectures would break the Java’s promise of “write once, run everywhere”. Happily for the programmers, the JMM specifies some guarantees that you may rely upon when designing multithreaded applications. Sticking to these guarantees helps a programmer to write multithreaded code that is stable and portable between various architectures.

The main notions of JMM are:

* **Actions**, these are inter-thread actions that can be executed by one thread and detected by another thread, like reading or writing variables, locking/unlocking monitors and so on
* **Synchronization actions**, a certain subset of actions, like reading/writing a *volatile*variable, or locking/unlocking a monitor
* **Program Order** (PO), the observable total order of actions inside a single thread
* **Synchronization Order** (SO), the total order between all synchronization actions — it has to be consistent with Program Order, that is, if two synchronization actions come one before another in PO, they occur in the same order in SO
* **synchronizes-with** (SW) relation between certain synchronization actions, like unlocking of monitor and locking of the same monitor (in another or the same thread)
* **Happens-before Order** — combines PO with SW (this is called *transitive closure* in set theory) to create a partial ordering of all actions between threads. If one action *happens-before* another, then the results of the first action are observable by the second action (for instance, write of a variable in one thread and read in another)
* **Happens-before consistency** — a set of actions is HB-consistent if every read observes either the last write to that location in the happens-before order, or some other write via data race
* **Execution** — a certain set of ordered actions and consistency rules between them

For a given program, we can observe multiple different executions with various outcomes. But if a program is **correctly synchronized**, then all of its executions appear to be **sequentially consistent**, meaning you can reason about the multithreaded program as a set of actions occurring in some sequential order. This saves you the trouble of thinking about under-the-hood reorderings, optimizations or data caching.

### Q10. What is a volatile field and what guarantees does the JMM hold for such field?

A *volatile* field has special properties according to the Java Memory Model (see Q9). The reads and writes of a *volatile* variable are synchronization actions, meaning that they have a total ordering (all threads will observe a consistent order of these actions). A read of a volatile variable is guaranteed to observe the last write to this variable, according to this order.

If you have a field that is accessed from multiple threads, with at least one thread writing to it, then you should consider making it *volatile*, or else there is a little guarantee to what a certain thread would read from this field.

Another guarantee for *volatile* is atomicity of writing and reading 64-bit values (*long* and *double*). Without a volatile modifier, a read of such field could observe a value partly written by another thread.

### Q11. Which of the following operations are atomic?

* writing to a non-*volatile* *int*;
* writing to a *volatile int*;
* writing to a non-*volatile long*;
* writing to a *volatile long*;
* incrementing a *volatile long*?

A write to an *int* (32-bit) variable is guaranteed to be atomic, whether it is *volatile* or not. A *long*(64-bit) variable could be written in two separate steps, for example, on 32-bit architectures, so by default, there is no atomicity guarantee. However, if you specify the *volatile* modifier, a *long*variable is guaranteed to be accessed atomically.

The increment operation is usually done in multiple steps (retrieving a value, changing it and writing back), so it is never guaranteed to be atomic, wether the variable is *volatile* or not. If you need to implement atomic increment of a value, you should use classes *AtomicInteger*, *AtomicLong* etc.

### Q12. What special guarantees does the JMM hold for final fields of a class?

JVM basically guarantees that *final* fields of a class will be initialized before any thread gets hold of the object. Without this guarantee, a reference to an object may be published, i.e. become visible, to another thread before all the fields of this object are initialized, due to reorderings or other optimizations. This could cause racy access to these fields.

This is why, when creating an immutable object, you should always make all its fields *final*, even if they are not accessible via getter methods.

### Q13. What is the meaning of a synchronized keyword in the definition of a method? Of a static method? Before a block?

The *synchronized* keyword before a block means that any thread entering this block has to acquire the monitor (the object in brackets). If the monitor is already acquired by another thread, the former thread will enter the *BLOCKED* state and wait until the monitor is released.

|  |  |
| --- | --- |
| 1  2  3 | **synchronized**(object) {  // ...  } |

A *synchronized* instance method has the same semantics, but the instance itself acts as a monitor.

|  |
| --- |
| **synchronized** **void** instanceMethod() {  // ...  } |

For a *static synchronized* method, the monitor is the *Class* object representing the declaring class.

|  |
| --- |
| **static** **synchronized** **void** staticMethod() {  // ...  } |

### Q14. If two threads call a synchronized method on different object instances simultaneously, could one of these threads block? What if the method is static?

If the method is an instance method, then the instance acts as a monitor for the method. Two threads calling the method on different instances acquire different monitors, so none of them gets blocked.

If the method is *static*, then the monitor is the *Class* object. For both threads, the monitor is the same, so one of them will probably block and wait for another to exit the *synchronized* method.

### Q15. What is the purpose of the *wait*, *notify* and *notifyAll* methods of the *Object* class?

A thread that owns the object’s monitor (for instance, a thread that has entered a *synchronized*section guarded by the object) may call *object.wait()* to temporarily release the monitor and give other threads a chance to acquire the monitor. This may be done, for instance, to wait for a certain condition.

When another thread that acquired the monitor fulfills the condition, it may call *object.notify()*or *object.notifyAll()* and release the monitor. The *notify* method awakes a single thread in the waiting state, and the *notifyAll* method awakes all threads that wait for this monitor, and they all compete for re-acquiring the lock.

The following *BlockingQueue* implementation shows how multiple threads work together via the *wait-notify* pattern. If we *put* an element into an empty queue, all threads that were waiting in the *take* method wake up and try to receive the value. If we *put* an element into a full queue, the *put* method *wait*s for the call to the *get* method. The *get* method removes an element and notifies the threads waiting in the *put* method that the queue has an empty place for a new item.

|  |
| --- |
| **public** **class** BlockingQueue<T> {    **private** List<T> queue = **new** LinkedList<T>();    **private** **int** limit = 10;    **public** **synchronized** **void** put(T item) {  **while** (queue.size() == limit) {  **try** {  wait();  } **catch** (InterruptedException e) {}  }  **if** (queue.isEmpty()) {  notifyAll();  }  queue.add(item);  }    **public** **synchronized** T take() **throws** InterruptedException {  **while** (queue.isEmpty()) {  **try** {  wait();  } **catch** (InterruptedException e) {}  }  **if** (queue.size() == limit) {  notifyAll();  }  **return** queue.remove(0);  }    } |

### Q16. Describe the conditions of deadlock, livelock, and starvation. Describe the possible causes of these conditions.

**Deadlock** is a condition within a group of threads that cannot make progress because every thread in the group has to acquire some resource that is already acquired by another thread in the group. The most simple case is when two threads need to lock both of two resources to progress, the first resource is already locked by one thread, and the second by another. These threads will never acquire a lock to both resources and thus will never progress.

**Livelock** is a case of multiple threads reacting to conditions, or events, generated by themselves. An event occurs in one thread and has to be processed by another thread. During this processing, a new event occurs which has to be processed in the first thread, and so on. Such threads are alive and not blocked, but still, do not make any progress because they overwhelm each other with useless work.

**Starvation** is a case of a thread unable to acquire resource because other thread (or threads) occupy it for too long or have higher priority. A thread cannot make progress and thus is unable to fulfill useful work.

### Q17. Describe the purpose and use-cases of the *fork/join*framework.

The fork/join framework allows parallelizing recursive algorithms. The main problem with parallelizing recursion using something like *ThreadPoolExecutor* is that you may quickly run out of threads because each recursive step would require its own thread, while the threads up the stack would be idle and waiting.

The fork/join framework entry point is the *ForkJoinPool* class which is an implementation of *ExecutorService*. It implements the work-stealing algorithm, where idle threads try to “steal” work from busy threads. This allows to spread the calculations between different threads and make progress while using fewer threads than it would require with a usual thread pool.

More information and code samples for the fork/join framework may be found in the article [“Guide to the Fork/Join Framework in Java”](https://www.baeldung.com/java-fork-join).

Question 1. What Is Countdownlatch In Java Concurrency?

Answer :CountDownLatch can be visualized as a latch that is released only after the given number of events occur. CountDownLatch is initialized with that count (given number of events).

Each time one of those events occur count is decremented, for that countdown() method is used. Thread(s) that are waiting for the latch to release (current count reaches zero due to invocations of the countDown()method) are blocked using await() method.

It is useful in the scenario when you want one or more threads to wait until one or more events being performed in other threads complete.

Question 2. What Is Cyclicbarrier In Java Concurrency?

Answer :CyclicBarrier is useful in scenarios where you want set of threads to wait for each other to reach a common barrier point. When each thread reaches the barrier (common point) you need to call await() method on the CyclicBarrier object. This will suspend the thread until all the thread also call the await() method on the same CyclicBarrier object.

Once all the specified threads have called await() method that will trip the barrier and all threads can resume operation.

The barrier is called cyclic because it can be re-used after the waiting threads are released.

Question 3. What Is The Difference Between A Countdownlatch And Cyclicbarrier?

Answer :When you are using a CountDownLatch, you specify the number of calls to the countdown() method when creating a CountDownLatch object. What this means is you can use CountDownLatch with only a single thread and using countdown() to decrement as and when the specified even occur.

When you are using CyclicBarrier you specify the number of threads that should call await() method in order to trip the barrier. That means if you have a CyclicBarrier initialized to 3 that means you should have at least 3 threads to call await().

CountDownLatch can't be reused, when count reaches zero it cannot be reset.

CyclicBarrier can be reused after the waiting threads are released.

Question 4. If A Countdownlatch Is Initialized With Some Count Let's Say 3 (new Countdownlatch(3)). Do We Need To Have 3 Threads For Countdown In That Case?

Answer :No. Same number of threads are not required. A CountDownLatch initialized to N can be used to make one thread wait until N threads have completed some action, or some action has been completed N times.

Question 5. What Is Phaser In Java Concurrency?

Answer :Phaser is more suitable for use where it is required to synchronize threads over one or more phases of activity. Though Phaser can be used to synchronize a single phase, in that case it acts more like a CyclicBarrier.

Phaser is reusable (like CyclicBarrier) and more flexible in usage.

The number of parties registered to synchronize on a phaser may vary over time. Tasks may be registered at any time (using methods register(), bulkRegister(int), or by specifying initial number of parties in the constructor). Tasks may also be optionally deregistered upon any arrival (using arriveAndDeregister()).

Question 6. What Is Exchanger In Java Concurrency?

Answer :Exchanger makes it easy for two threads to exchange data between themselves.

Exchanger provides a synchronization point at which two threads can pair and swap elements. Exchanger waits until two separate threads call its exchange() method. When two threads have called the exchange() method, Exchanger will swap the objects presented by the threads.

Question 7. What Is Semaphore In Java Concurrency?

Answer :The Semaphore class present in java.util.concurrent package is a counting semaphore in which a semaphore, conceptually, maintains a set of permits. Thread that wants to access the shared resource tries to acquire a permit using acquire() method. At that time if the Semaphore's count is greater than zero thread will acquire a permit and Semaphore's count will be decremented by one. If Semaphore's count is zero, when thread calls acquire() method, then the thread will be blocked until a permit is available. When thread is done with the shared resource access, it can call the release() method to release the permit. That results in the Semaphore's count incremented by one.

Question 8. What Is Reentrantlock In Java?

Answer :ReentrantLock is a concrete implementation of the Lock interface which is present injava.util.concurrent.locks package.

Every object created in Java has one mutually exclusive lock associated with it. When you are using synchronized you are using that lock implicitly (with no other feature) whereas when you are using any of the lock implementation (like Reentrant lock) you are using that lock explicitly. Which means there are methods like lock() to acquire the lock and unlock() to release the lock. Along with that ReentrantLock provides many other features like fairness, ability to interrupt and a thread waiting for a lock only for a specified period.

Question 9. What Is The Difference Between Reentrantlock And Synchronized?

Answer :When you use a synchronized block or method you just need to write synchronized keyword (and provide associated object) acquiring lock and releasing it is done implicitly.

With ReentrantLock acquiring and releasing lock is done by user using lock() and unlock() methods.

Synchronized forces all lock acquisition and release to occur in a block-structured way which means when multiple locks are acquired they must be released in the opposite order, and all locks must be released in the same lexical scope in which they were acquired.

ReentrantLock provides more flexibility, it allows a lock to be acquired and released in different scopes, and allowing multiple locks to be acquired and released in any order.

ReentrantLock provides additional functionality over the use of synchronized methods and statements by providing an option for fairness, providing a non-blocking attempt to acquire a lock (tryLock()), an attempt to acquire the lock that can be interrupted (lockInterruptibly(), and an attempt to acquire the lock that can timeout (tryLock(long, TimeUnit)).

Question 10. Why Is It Named Reentrantlock?

Answer :It is called ReentrantLock as there is an acquisition count associated with the lock which means when you use lock() method to acquire a lock and you get it then the acquisition count is 1.

A Reentrant lock will also allow the lock holder to enter another block of code with the same lock object as thread already owned it. In that case, if a thread that holds the lock acquires it again, the acquisition count is incremented and the lock then needs to be released twice to truly release the lock.

Question 11. What Is Readwritelock In Java?

Answer :In a multi-threading application multiple reads can occur simultaneously for a shared resource. It is only when multiple writes happen simultaneously or intermix of read and write that there is a chance of writing the wrong value or reading the wrong value.

ReadWriteLock uses the same idea in order to boost the performance by having separate pair of locks. A ReadWriteLock maintains a pair of associated locks -

One for read-only operations; and One for writing.

The read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive.

Question 12. What Is Reentrantreadwritelock In Java Concurrency?

Answer :ReentrantReadWriteLock is an implementation of the ReadWriteLock interface which provides a pair of read-write lock.

To get a read lock you need to use - rw.readLock().lock();

To get a write lock you need to use - rw.writeLock().lock();

Where rw is an object of ReentrantReadWriteLock class.

ReentrantReadWriteLock also allows downgrading from the write lock to a read lock. You can first acquire a write lock, then the read lock and then release the write lock.

Question 13. How Readwritelock Can Help In Reducing Contention Among Multiple Threads? Or What Is Benefit Of Using Readwritelock In Java?

Answer :

ReadWriteLock provides separate set of locks for reading and writing operations. Where read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive.

So read operations are not mutually exclusive. It exploits the fact that while only a single thread at a time (a writer thread) can modify the shared data, in many cases any number of threads can concurrently read the data (hence reader threads).

Thus in the applications where reads are more than writes or duration of reads is more the thread contention will be less as read lock is shared by many thread rather than being mutually exclusive. So you won't have a situation where only one thread is reading and other threads are waiting.

Question 14. What Is Concurrenthashmap In Java?

Answer :

ConcurrentHashMap is also a hash based map like HashMap, how it differs is the locking strategy used by ConcurrentHashMap. Unlike HashTable (or synchronized HashMap) it doesn't synchronize every method on a common lock. ConcurrentHashMap uses separate lock for separate buckets thus locking only a portion of the Map.

That way ConcurrentHashMap depite being a thread safe alternative to HashTable gives much better performance.

Question 15. What Is Lock Striping In Concurrent Programming?

Answer :

The concept of lock striping is to have separate locks for a portion of a data structure where each lock is locking on a variable sized set of independent objects.

That's how ConcurrentHashMap in Java provides synchronization. By default ConcurrentHashMap has 16 buckets and each bucket has its own lock so there are 16 locks too. So the threads which are accessing keys in separate buckets can access them simultaneously.

Question 16. What Is The Difference Between Hashmap And Concurrenthashmap In Java?

Answer :

ConcurrentHashMap is thread safe and fit for use in a multi-threaded environment whereas HashMap is not thread safe.

HashMap can be synchronized using the Collections.synchronizedMap() method but that synchronizes all the methods of the HashMap and effectively reduces it to a data structure where one thread can enter at a time.

In ConcurrentHashMap synchronization is done a little differently. Rather than locking every method on a common lock, ConcurrentHashMap uses separate lock for separate buckets thus locking only a portion of the Map.

Question 17. Why Concurrenthashmap Is Faster Than Hashtable In Java?

Answer :

In HashTable each method is synchronized on a single lock which means at any given time only one thread can enter any method.

ConcurrentHashMap uses separate lock for separate buckets thus locking only a portion of the Map. By default there are 16 buckets and also separate locks for separate buckets. So the default concurrency level is 16. Thus theoretically at any given time 16 threads can access separate buckets without blocking which improves the performance of the ConcurrentHashMap.

In ConcurrentHashMap performance is further improved by providing read access concurrently without any blocking. Retrieval operations (including get) generally do not block, so may overlap with update operations (including put and remove).

Question 18. What Is Copyonwritearraylist In Java?

Answer :

CopyOnWriteArrayList is also an implementation of the List interface but it is a thread safe variant. This thread safety is achieved by making a fresh copy of the underlying array with every mutative operations (add, set, and so on).

Using CopyOnWriteArrayList provides better performance in scenarios where there are more iterations of the list than mutations.

Question 19. What Is The Difference Between Arraylist And Copyonwritearraylist In Java?

Answer :

ArrayList is not thread-safe whereas CopyOnWriteArrayList is thread-safe and fit for use in multi-threaded environment.

Iterator returned by ArrayList is fail-fast. Iterator returned by CopyOnWriteArrayList is fail-safe.

Performance wise ArrayList is faster as it is not synchronized and there is no added burden of thread-safety. CopyOnWriteArrayList is comparatively slower and if there are lots of writes by various threads that will degrade the performance of the CopyOnwriteArrayList as there will be copies made per mutation.

Question 20. What Is Copyonwritearrayset In Java?

Answer :

CopyOnWriteArraySet is a thread-safe collection and it internally uses CopyOnWriteArrayList for all of its operations.

Since it uses CopyOnWriteArrayList internally so thread-safety is achieved in the same way in CopyOnwriteArraySet as in CopyOnWriteArrayList - all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array.

The iterator returned by CopyOnwriteArraySet is fail-safe which means any structural modification made to the CopyOnwriteArraySet won't throw ConcurrentModificationException.

Question 21. What Is Concurrentskiplistmap In Java?

Answer :

ConcurrentSkipListMap implements ConcurrentNavigableMap and it is a sorted map just like TreeMap with the added feature of being concurrent.

ConcurrentSkipListMap 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.

Question 22. What Is Concurrentskiplistset In Java?

Answer :

ConcurrentSkipListSet implements NavigableSet and it is a sorted set just like TreeSet with added feature of being concurrent.

The elements of the set are kept sorted according to their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.

Question 23. What Is Concurrentlinkedqueue In Java?

Answer :

ConcurrentLinkedQueue is an unbounded thread-safe queue which stores its elements as linked nodes. This queue orders elements FIFO (first-in-first-out).

It doesn't block operations as it is done in the implementations of BlockingQueue interface like ArrayBlockingQueue.

Question 24. What Is Concurrentlinkeddequeue In Java?

Answer :

ConcurrentLinkedDeque is an unbounded thread-safeDeque which stores its elements as linked nodes. Since it implements deque interface ConcurrentLinkedDequesupports element insertion and removal at both ends.

ConcurrentLinkedDequeue is thread safe and it doesn't block operations.

Question 25. What Do You Mean By Non-blocking Algorithm/data Structure?

Answer :

An algorithm is called non-blocking if it doesn't block threads in such a way that only one thread has access to the data structure and all the other threads are waiting. Same way failure of any thread in a non-blocking algorithm doesn't mean failure or suspension of other threads.

Implementation of non-blocking data structures in Java like atomic variables or ConcurrentLinkedQueue use an atomic read-modify-write kind of instruction based on compare-and-swap.

Question 26. What Is Busy Spinning? When Will You Use Busy Spinning As Waiting Strategy?

Answer :

An algorithm is called non-blocking if it doesn't block threads in such a way that only one thread has access to the data structure and all the other threads are waiting. Same way failure of any thread in a non-blocking algorithm doesn't mean failure or suspension of other threads.

Implementation of non-blocking data structures in Java like atomic variables or ConcurrentLinkedQueue use an atomic read-modify-write kind of instruction based on compare-and-swap.

Question 27. What Is Blockingqueue In Java Concurrency?

Answer :

BlockingQueueinterface is added in Java 5 with in the java.util.concurrent package.

BlockingQueue is a queue that can block the operations. Which means BlockingQueue supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

BlockingQueue provides following blocking methods -

put(E e) - Inserts the specified element into this queue, waiting if necessary for space to become available.

take() - Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

Question 28. What Is Blocking Method In Java?

Answer :

Methods which want to execute the task assigned without relinquishing control to other thread are called blocking methods.

A very relevant example of blocking methods, which most of you would have encountered is read() method of theInputStream class. This method blocks until input data is available, the end of the stream is detected, or an exception is thrown.

Question 29. What Is Arrayblockingqueue In Java Concurrency?

Answer :

Methods which want to execute the task assigned without relinquishing control to other thread are called blocking methods.

A very relevant example of blocking methods, which most of you would have encountered is read() method of theInputStream class. This method blocks until input data is available, the end of the stream is detected, or an exception is thrown.

Question 30. What Is Linkedblockingqueue In Java Concurrency?

Answer :

LinkedBlockingQueue is an implementation of BlockingQueue interface.

LinkedBlockingQueue internally uses linked nodes to store elements. It is optionally bounded and that's where it differs from ArrayBlockingQueue which is bounded.

Question 31. What Is Priorityblockingqueue In Java Concurrency?

Answer :

PriorityBlockingQueue class implements the BlockingQueue interface. The elements of the PriorityBlockingQueue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which of the following constructor is used.

PriorityBlockingQueue() - Creates a PriorityBlockingQueue with the default initial capacity (11) that orders its elements according to their natural ordering.

PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator) - Creates a PriorityBlockingQueue with the specified initial capacity that orders its elements according to the specified comparator.

Question 32. What Is Synchronous Queue In Java?

Answer :

SynchronousQueue is an implementation of the BlockingQueue interface. SynchronousQueue does not have any internal capacity, not even a capacity of one. In SynchronousQueue each insert operation must wait for a corresponding remove operation by another thread, and vice versa.

If you put an element in SynchronousQueue using put() method it will wait for another thread to receive it, you can't put any other element in the SynchronousQueue as it is blocked.

Question 33. What Is Delayqueue In Java Concurrency?

Answer :

DelayQueue is an unbounded implementation of BlockingQueue interface. DelayQueue can store elements of type Delayed only and an element can only be retrieved from DelayQueue when its delay has expired.

When you implement Delayed interface two methods have to be implementedgetDelay(TimeUnit unit) and compareTo(T o).

getDelay(TimeUnit unit) - Returns the remaining delay associated with this object, in the given time unit.

Question 34. What Is Transferqueue In Java?

Answer :

TransferQueue interface, added in Java 7, extends BlockingQueue interface. The extra functionality provided by TransferQueue interface is that it provides blocking method which will wait until other thread receives your element.

That's how it differs from BlockingQueue where you can only put element into queue or retrieve element from queue and block if queue is full (while you are putting elements) or block if queue is empty (while you are retrieving elements).

TransferQueue has a blocking method transfer(E e) which will ensure that the element is transferred to the consumer, it will wait if required to do so.

Question 35. What Is Linkedtransferqueue In Java?

Answer :

LinkedTransferQueue, is an implementation of the TransferQueue. It is an unbounded queue and stores elements as linked nodes.

Question 36. What Is Blockingdeque In Java Concurrency?

Answer :

BlockingDeque interface (added in Java 6) is a Deque that provides additional support for blocking operations. Blocking methods of BlockingDeque interface come in four forms.

Throw exception - Methods falling in this category will throw exception if blocked.

Return special value - This type of methods will return some value if need to wait, like false.

Blocks - This type of methods will wait if necessary for space to become available.

Times out - This type of methods will block for only a given maximum time limit before giving up.

BlockingDeque is thread safe, does not permit null elements, and may (or may not) be capacity-constrained.

Question 37. What Is Linkedblockingdeque In Java?

Answer :

LinkedBlockingDeque is an implementation of the BlockingDeque interface and it was added in Java 6. LinkedBlockingDeque is an optionally bounded deque and it stores its elements as linked nodes.

Question 38. What Is Executor In Java Concurrency?

Answer :

The concurrent API has a feature called executors that provides an alternative to managing threads through the Thread class. At the core of the executors is the Executor interface - An object of type Executor can execute runnable tasks. An Executor is normally used instead of explicitly creating threads.

For example If r is a Runnable object, and e is an Executor object you can replace

(new Thread(r)).start();

with

e.execute(r);

The Executor interface provides a single method, execute -

void execute(Runnable command)

Question 39. What Is Executorservice In Java Concurrency?

Answer :

ExecutorService interface extends Executor interface and provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.

ExecutorService has more versatile submit method. Like execute, submit accepts Runnable objects, but also accepts Callable objects, which allow the task to return a 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.

Question 40. Name Any Class That Implements Executor Or Executorservice Interface?

Answer :

In the Java concurrency there are three pre defined executor classes that implement the Executor and ExecutorService interface.

ThreadPoolExecutor - Implements the Executor and ExecutorService interfaces and executes the submitted task using one of the pooled thread.

ScheduledThreadPoolExecutor - It extends ThreadPoolExecutor and also implements the ScheduledExecutorService. This class schedule commands to run after a given delay, or to execute periodically.

ForkJoinPool - It implements the Executor and ExecutorService interfaces and is used by the Fork/Join Framework.

Question 41. What Is Difference Between Submit() And Execute() Method Of Executor And Executorservice In Java?

Answer :

execute() method is provided by Executor interface where as submit() method is provided by ExecutorService.

execute() method only takes Runnable as argument - execute(Runnable command) and does not return any value.

ExecutorService has more versatile submit() method. submit() method is overloaded and accepts both Runnable objects and Callable objects, submit also allows the task to return a value (an object of type Future). The Future's get method will return the given result upon successful completion.

Question 42. How Can I Immediately Block A Thread Even If I Am Using Submit Method And Using A Callable Object?

Answer :

If you would like to immediately block waiting for a task, you can use constructions of the form

result = exec.submit(aCallable).get();

Question 43. What Will Happen If Submit Method Can’t Schedule A Task For Execution?

Answer :

It will throw RejectedExecutionException.

Question 44. How To Shut Down An Executorservice?

Answer :An ExecutorService can be shut down, which will cause it to reject new tasks. Two different methods are provided for shutting down an ExecutorService.

The shutdown() method will allow previously submitted tasks to execute before terminating, while the shutdownNow() method prevents waiting tasks from starting and attempts to stop currently executing tasks. Upon termination, an executor has no tasks actively executing, no tasks awaiting execution, and no new tasks can be submitted.

Question 45. What Is A Scheduledexecutorservice?

Answer :ScheduledExecutorService extends ExecutorService and provides methods that can schedule commands to run after a given delay, or to execute periodically.

It has methods that execute a Runnable or Callable task after a specified delay.

schedule(Callable<V> callable, long delay, TimeUnit unit) - Creates and executes a ScheduledFuture that becomes enabled after the given delay.

schedule(Runnable command, long delay, TimeUnit unit) - Creates and executes a one-shot action that becomes enabled after the given delay.

Question 46. What Is Executors Class?

Answer :Executors class provide factory and utility methods for Executors framework classes like Executor, ExecutorService, ScheduledExecutorService, ThreadFactory, and Callable.

Though you can use ThreadPoolExecutor and ScheduledThreadPoolExecutor directly, but the best way to get an executor is to use one of the static factory methods provided by the Executors utility class.

Some of the factory methods -

static ExecutorService newCachedThreadPool() - Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

static ExecutorService newFixedThreadPool(int numThreads) - Creates a thread pool that reuses a fixed number of threads.

static ScheduledExecutorService newScheduledThreadPool(int numThreads) - Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

newSingleThreadExecutor() - Creates an Executor that uses a single worker thread operating off an unbounded queue.

As example -

ExecutorService ex = Executors.newFixedThreadPool(2);

Question 47. What Is Threadpool In Java?

Answer :In a large scale application if each task uses its own thread then allocating and deallocating many thread objects creates a significant memory management overhead.

Thread pool as the name suggests provides a set of threads, any task which has to be executed get a thread from this pool.

// creating executor with pool of 2 threads

ExecutorService ex = Executors.newFixedThreadPool(2);

// running 6 tasks

ex.execute(new Task());

ex.execute(new Task());

ex.execute(new Task());

ex.execute(new Task());

ex.execute(new Task());

ex.execute(new Task());

//shutting down the executor service

ex.shutdown();

Even if we are running 6 tasks here, all these tasks would be run using the 2 threads from the pool.

Question 48. How To Construct A Thread Pool With 2 Threads That Executes Some Tasks That Return A Value?

Answer :You can create a fixed thread pool using the newFixedThreadPool() method of the Executors class.

// creating executor with pool of 2 threads

ExecutorService ex = Executors.newFixedThreadPool(2);

// running tasks

Future f1 = ex.submit(new Task());

Future f2 = ex.submit(new Task());

try {

// getting the future value

System.out.println("Future f1 " + f1.get());

System.out.println("Future f1 " + f1.get());

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (ExecutionException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

ex.shutdown();

Question 49. What Is Callable And Future In Java Concurrency?

Answer :Callable, an interface, was added in Java 5. It allows you to define a task to be completed by a thread asynchronously. The Callable interface has a call() method, since it is a generic interface so it can return any value (Object, String, Integer etc.) based on how it is initialized. Main feature of the call() method provided by Callable interface is that it can return value.

Future interface - A Future represents the result of an asynchronous computation. When you submit a callable task using the submit() method of the ExecutorService, Future object is returned.

Future provides methods to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation.

get() - get() method retrieves the result of the computation, blocking if necessary for the computation to complete.

Question 50. What Is Atomic Variable In Java?

Answer :In Java concurrency classes like AtomicInteger, AtomicLong are provided with a int, long value respectively that may be updated atomically.

These atomic variable classes in Java concurrency like AtomicInteger, AtomicLong uses non-blocking algorithm. These non-blocking algorithms use low-level atomic machine instructions such as compare-and-swap instead of locks to ensure data integrity under concurrent access.