Module java.base

Package java.util.concurrent

```
package java.util.concurrent
```

Utility classes commonly useful in concurrent programming. This package includes a few small standardized extensible frameworks, as well as some classes that provide useful functionality and are otherwise tedious or difficult to implement. Here are brief descriptions of the main components. See also the `java.util.concurrent.locks` and `java.util.concurrent.atomic` packages.

Executors

Interfaces. `Executor` is a simple standardized interface for defining custom thread-like subsystems, including thread pools, asynchronous I/O, and lightweight task frameworks. Depending on which concrete Executor class is being used, tasks may execute in a newly created thread, an existing task-execution thread, or the thread calling `execute`, and may execute sequentially or concurrently. `ExecutorService` provides a more complete asynchronous task execution framework. An ExecutorService manages queuing and scheduling of tasks, and allows controlled shutdown. The `ScheduledExecutorService` subinterface and associated interfaces add support for delayed and periodic task execution. ExecutorServices provide methods arranging asynchronous execution of any function expressed as `Callable`, the result-bearing analog of `Runnable`. A `Future` returns the results of a function, allows determination of whether execution has completed, and provides a means to cancel execution. A `RunnableFuture` is a `Future` that possesses a `run` method that upon execution, sets its results.

Implementations. Classes `ThreadPoolExecutor` and `ScheduledThreadPoolExecutor` provide tunable, flexible thread pools. The `Executors` class provides factory methods for the most common kinds and configurations of Executors, as well as a few utility methods for using them. Other utilities based on `Executors` include the concrete class `FutureTask` providing a common extensible implementation of Futures, and `ExecutorCompletionService`, that assists in coordinating the processing of groups of asynchronous tasks.

Class `ForkJoinPool` provides an Executor primarily designed for processing instances of `ForkJoinTask` and its subclasses. These classes employ a work-stealing scheduler that attains high throughput for tasks conforming to restrictions that often hold in computation-intensive parallel processing.

Queues

The `ConcurrentLinkedQueue` class supplies an efficient scalable thread-safe non-blocking FIFO queue. The `ConcurrentLinkedDeque` class is similar, but additionally supports the `Deque` interface.

Five implementations in `java.util.concurrent` support the extended `BlockingQueue` interface, that defines blocking versions of put and take: `LinkedBlockingQueue`, `ArrayBlockingQueue`, `SynchronousQueue`, `PriorityBlockingQueue`, and `DelayQueue`. The different classes cover the most common usage contexts for producer-consumer, messaging, parallel tasking, and related concurrent designs.

Extended interface `TransferQueue`, and implementation `LinkedTransferQueue` introduce a synchronous `transfer` method (along with related features) in which a producer may optionally block awaiting its consumer.

The BlockingDeque interface extends BlockingQueue to support both FIFO and LIFO (stack-based) operations. Class LinkedBlockingDeque provides an implementation.

Timing

The TimeUnit class provides multiple granularities (including nanoseconds) for specifying and controlling time-out based operations. Most classes in the package contain operations based on time-outs in addition to indefinite waits. In all cases that time-outs are used, the time-out specifies the minimum time that the method should wait before indicating that it timed-out. Implementations make a "best effort" to detect time-outs as soon as possible after they occur. However, an indefinite amount of time may elapse between a time-out being detected and a thread actually executing again after that time-out. All methods that accept timeout parameters treat values less than or equal to zero to mean not to wait at all. To wait "forever", you can use a value of Long.MAX_VALUE.

Synchronizers

Five classes aid common special-purpose synchronization idioms.

- Semaphore is a classic concurrency tool.
- CountDownLatch is a very simple yet very common utility for blocking until a given number of signals, events, or conditions hold.
- A CyclicBarrier is a resettable multiway synchronization point useful in some styles of parallel programming.
- A Phaser provides a more flexible form of barrier that may be used to control phased computation among multiple threads.
- An Exchanger allows two threads to exchange objects at a rendezvous point, and is useful in several pipeline designs.

Concurrent Collections

Besides Queues, this package supplies Collection implementations designed for use in multithreaded contexts: ConcurrentHashMap, ConcurrentSkipListMap, ConcurrentSkipListSet, CopyOnWriteArrayList, and CopyOnWriteArraySet. When many threads are expected to access a given collection, a ConcurrentHashMap is normally preferable to a synchronized HashMap, and a ConcurrentSkipListMap is normally preferable to a synchronized TreeMap. A CopyOnWriteArrayList is preferable to a synchronized ArrayList when the expected number of reads and traversals greatly outnumber the number of updates to a list.

The "Concurrent" prefix used with some classes in this package is a shorthand indicating several differences from similar "synchronized" classes. For example java.util.Hashtable and Collections.synchronizedMap(new HashMap()) are synchronized. But ConcurrentHashMap is "concurrent". A concurrent collection is thread-safe, but not governed by a single exclusion lock. In the particular case of ConcurrentHashMap, it safely permits any number of concurrent reads as well as a large number of concurrent writes. "Synchronized" classes can be useful when you need to prevent all access to a collection via a single lock, at the expense of poorer scalability. In other cases in which multiple threads are expected to access a common collection, "concurrent" versions are normally preferable. And unsynchronized collections are preferable when either collections are unshared, or are accessible only when holding other locks.

Most concurrent Collection implementations (including most Queues) also differ from the usual java.util conventions in that their Iterators and Spliterators provide *weakly consistent* rather than fast-fail traversal:

- they may proceed concurrently with other operations
- they will never throw ConcurrentModificationException
- they are guaranteed to traverse elements as they existed upon construction exactly once, and may (but are not guaranteed to) reflect any modifications subsequent to

construction.

Memory Consistency Properties

Chapter 17 of *The Java Language Specification* defines the *happens-before* relation on memory operations such as reads and writes of shared variables. The results of a write by one thread are guaranteed to be visible to a read by another thread only if the write operation *happens-before* the read operation. The `synchronized` and `volatile` constructs, as well as the `Thread.start()` and `Thread.join()` methods, can form *happens-before* relationships. In particular:

- Each action in a thread *happens-before* every action in that thread that comes later in the program's order.
- An unlock (`synchronized` block or method exit) of a monitor *happens-before* every subsequent lock (`synchronized` block or method entry) of that same monitor. And because the *happens-before* relation is transitive, all actions of a thread prior to unlocking *happen-before* all actions subsequent to any thread locking that monitor.
- A write to a `volatile` field *happens-before* every subsequent read of that same field. Writes and reads of `volatile` fields have similar memory consistency effects as entering and exiting monitors, but do *not* entail mutual exclusion locking.
- A call to `start` on a thread *happens-before* any action in the started thread.
- All actions in a thread *happen-before* any other thread successfully returns from a `join` on that thread.

The methods of all classes in `java.util.concurrent` and its subpackages extend these guarantees to higher-level synchronization. In particular:

- Actions in a thread prior to placing an object into any concurrent collection *happen-before* actions subsequent to the access or removal of that element from the collection in another thread.
- Actions in a thread prior to the submission of a `Runnable` to an `Executor` *happen-before* its execution begins. Similarly for `Callables` submitted to an `ExecutorService`.
- Actions taken by the asynchronous computation represented by a `Future` *happen-before* actions subsequent to the retrieval of the result via `Future.get()` in another thread.
- Actions prior to "releasing" synchronizer methods such as `Lock.unlock`, `Semaphore.release`, and `CountDownLatch.countDown` *happen-before* actions subsequent to a successful "acquiring" method such as `Lock.lock`, `Semaphore.acquire`, `Condition.await`, and `CountDownLatch.await` on the same synchronizer object in another thread.
- For each pair of threads that successfully exchange objects via an `Exchanger`, actions prior to the `exchange()` in each thread *happen-before* those subsequent to the corresponding `exchange()` in another thread.
- Actions prior to calling `CyclicBarrier.await` and `Phaser.awaitAdvance` (as well as its variants) *happen-before* actions performed by the barrier action, and actions performed by the barrier action *happen-before* actions subsequent to a successful return from the corresponding `await` in other threads.

See *Java Language Specification*:

17.4.5 Happens-before Order

Since:

1.5

Related Packages

Package	Description
java.util	Contains the collections framework, some internationalization support classes, a service loader, properties, random number

	generation, string parsing and scanning classes, base64 encoding and decoding, a bit array, and several miscellaneous utility classes.
java.util.concurrent.atomic	A small toolkit of classes that support lock-free thread-safe programming on single variables.
java.util.concurrent.locks	Interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors.

All Classes and Interfaces Interfaces Classes Enum Classes Exception Classes

Class	Description
AbstractExecutorService	Provides default implementations of `ExecutorService` execution methods.
ArrayBlockingQueue<E>	A bounded blocking queue backed by an array.
BlockingDeque<E>	A `Deque` that additionally supports blocking operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element.
BlockingQueue<E>	A `Queue` that additionally 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.
BrokenBarrierException	Exception thrown when a thread tries to wait upon a barrier that is in a broken state, or which enters the broken state while the thread is waiting.
Callable<V>	A task that returns a result and may throw an exception.
CancellationException	Exception indicating that the result of a value-producing task, such as a `FutureTask`, cannot be retrieved because the task was cancelled.
CompletableFuture<T>	A `Future` that may be explicitly completed (setting its value and status), and may be used as a `CompletionStage`, supporting dependent functions and actions that trigger upon its completion.

CompletableFuture.AsynchronousCompletion	A marker interface identifying asynchronous tasks produced by `async` methods.
CompletionException	Exception thrown when an error or other exception is encountered in the course of completing a result or task.
CompletionService<V>	A service that decouples the production of new asynchronous tasks from the consumption of the results of completed tasks.
CompletionStage<T>	A stage of a possibly asynchronous computation, that performs an action or computes a value when another CompletionStage completes.
ConcurrentHashMap<K,V>	A hash table supporting full concurrency of retrievals and high expected concurrency for updates.
ConcurrentHashMap.KeySetView<K,V>	A view of a ConcurrentHashMap as a `Set` of keys, in which additions may optionally be enabled by mapping to a common value.
ConcurrentLinkedDeque<E>	An unbounded concurrent deque based on linked nodes.
ConcurrentLinkedQueue<E>	An unbounded thread-safe queue based on linked nodes.
ConcurrentMap<K,V>	A `Map` providing thread safety and atomicity guarantees.
ConcurrentNavigableMap<K,V>	A `ConcurrentMap` supporting `NavigableMap` operations, and recursively so for its navigable sub-maps.
ConcurrentSkipListMap<K,V>	A scalable concurrent `ConcurrentNavigableMap` implementation.
ConcurrentSkipListSet<E>	A scalable concurrent `NavigableSet` implementation based on a `ConcurrentSkipListMap`.
CopyOnWriteArrayList<E>	A thread-safe variant of `ArrayList` in which all mutative operations (`add`, `set`, and so on) are implemented by making a fresh copy of the underlying array.
CopyOnWriteArraySet<E>	A `Set` that uses an internal `CopyOnWriteArrayList` for all of its operations.

CountDownLatch	A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.
CountedCompleter<T>	A `ForkJoinTask` with a completion action performed when triggered and there are no remaining pending actions.
CyclicBarrier	A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.
Delayed	A mix-in style interface for marking objects that should be acted upon after a given delay.
DelayQueue<E extends **Delayed**>	An unbounded blocking queue of `Delayed` elements, in which an element can only be taken when its delay has expired.
Exchanger<V>	A synchronization point at which threads can pair and swap elements within pairs.
ExecutionException	Exception thrown when attempting to retrieve the result of a task that aborted by throwing an exception.
Executor	An object that executes submitted `Runnable` tasks.
ExecutorCompletionService<V>	A `CompletionService` that uses a supplied `Executor` to execute tasks.
Executors	Factory and utility methods for `Executor`, `ExecutorService`, `ScheduledExecutorService`, `ThreadFactory`, and `Callable` classes defined in this package.
ExecutorService	An `Executor` that provides methods to manage termination and methods that can produce a `Future` for tracking progress of one or more asynchronous tasks.
Flow	Interrelated interfaces and static methods for establishing flow-controlled components in which `Publishers` produce items consumed by one or more `Subscribers`, each managed by a `Subscription`.
Flow.Processor<T,R>	A component that acts as both a Subscriber and Publisher.

Flow.Publisher<T>	A producer of items (and related control messages) received by Subscribers.
Flow.Subscriber<T>	A receiver of messages.
Flow.Subscription	Message control linking a `Flow.Publisher` and `Flow.Subscriber`.
ForkJoinPool	An `ExecutorService` for running `ForkJoinTask`s.
ForkJoinPool.ForkJoinWorkerThreadFactory	Factory for creating new `ForkJoinWorkerThread`s.
ForkJoinPool.ManagedBlocker	Interface for extending managed parallelism for tasks running in `ForkJoinPool`s.
ForkJoinTask<V>	Abstract base class for tasks that run within a `ForkJoinPool`.
ForkJoinWorkerThread	A thread managed by a `ForkJoinPool`, which executes `ForkJoinTask`s.
Future<V>	A `Future` represents the result of an asynchronous computation.
FutureTask<V>	A cancellable asynchronous computation.
LinkedBlockingDeque<E>	An optionally-bounded blocking deque based on linked nodes.
LinkedBlockingQueue<E>	An optionally-bounded blocking queue based on linked nodes.
LinkedTransferQueue<E>	An unbounded `TransferQueue` based on linked nodes.
Phaser	A reusable synchronization barrier, similar in functionality to `CyclicBarrier` and `CountDownLatch` but supporting more flexible usage.
PriorityBlockingQueue<E>	An unbounded blocking queue that uses the same ordering rules as class `PriorityQueue` and supplies blocking retrieval operations.
RecursiveAction	A recursive resultless `ForkJoinTask`.
RecursiveTask<V>	A recursive result-bearing `ForkJoinTask`.
RejectedExecutionException	Exception thrown by an `Executor` when a task cannot be accepted for execution.
RejectedExecutionHandler	A handler for tasks that cannot be executed by a `ThreadPoolExecutor`.

RunnableFuture<V>	A `Future` that is `Runnable`.
RunnableScheduledFuture<V>	A `ScheduledFuture` that is `Runnable`.
ScheduledExecutorService	An `ExecutorService` that can schedule commands to run after a given delay, or to execute periodically.
ScheduledFuture<V>	A delayed result-bearing action that can be cancelled.
ScheduledThreadPoolExecutor	A `ThreadPoolExecutor` that can additionally schedule commands to run after a given delay, or to execute periodically.
Semaphore	A counting semaphore.
SubmissionPublisher<T>	A `Flow.Publisher` that asynchronously issues submitted (non-null) items to current subscribers until it is closed.
SynchronousQueue<E>	A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa.
ThreadFactory	An object that creates new threads on demand.
ThreadLocalRandom	A random number generator (with period 2^{64}) isolated to the current thread.
ThreadPoolExecutor	An `ExecutorService` that executes each submitted task using one of possibly several pooled threads, normally configured using `Executors` factory methods.
ThreadPoolExecutor.AbortPolicy	A handler for rejected tasks that throws a `RejectedExecutionException`.
ThreadPoolExecutor.CallerRunsPolicy	A handler for rejected tasks that runs the rejected task directly in the calling thread of the `execute` method, unless the executor has been shut down, in which case the task is discarded.
ThreadPoolExecutor.DiscardOldestPolicy	A handler for rejected tasks that discards the oldest unhandled request and then retries `execute`, unless the executor is shut down, in which case the task is discarded.
ThreadPoolExecutor.DiscardPolicy	A handler for rejected tasks that silently discards the rejected task.

TimeoutException	Exception thrown when a blocking operation times out.
TimeUnit	A `TimeUnit` represents time durations at a given unit of granularity and provides utility methods to convert across units, and to perform timing and delay operations in these units.
TransferQueue<E>	A `BlockingQueue` in which producers may wait for consumers to receive elements.