java.util.concurrent

Class Semaphore

java.lang.Object
  └ **java.util.concurrent.Semaphore**

All Implemented Interfaces:
Serializable

```
public class Semaphore
extends Object
implements Serializable
```

A counting semaphore. Conceptually, a semaphore maintains a set of permits. Each `acquire()` blocks if necessary until a permit is available, and then takes it. Each `release()` adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the `Semaphore` just keeps a count of the number available and acts accordingly.

Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource. For example, here is a class that uses a semaphore to control access to a pool of items:

```
class Pool {
  private static final MAX_AVAILABLE = 100;
  private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);

  public Object getItem() throws InterruptedException {
    available.acquire();
    return getNextAvailableItem();
  }

  public void putItem(Object x) {
    if (markAsUnused(x))
      available.release();
  }

  // Not a particularly efficient data structure; just for demo

  protected Object[] items = ... whatever kinds of items being managed
  protected boolean[] used = new boolean[MAX_AVAILABLE];

  protected synchronized Object getNextAvailableItem() {
    for (int i = 0; i < MAX_AVAILABLE; ++i) {
      if (!used[i]) {
         used[i] = true;
         return items[i];
      }
    }
    return null; // not reached
  }
```

```
   protected synchronized boolean markAsUnused(Object item) {
     for (int i = 0; i < MAX_AVAILABLE; ++i) {
       if (item == items[i]) {
          if (used[i]) {
            used[i] = false;
            return true;
          } else
            return false;
       }
     }
     return false;
   }

 }
```

Before obtaining an item each thread must acquire a permit from the semaphore, guaranteeing that an item is available for use. When the thread has finished with the item it is returned back to the pool and a permit is returned to the semaphore, allowing another thread to acquire that item. Note that no synchronization lock is held when `acquire()` is called as that would prevent an item from being returned to the pool. The semaphore encapsulates the synchronization needed to restrict access to the pool, separately from any synchronization needed to maintain the consistency of the pool itself.

A semaphore initialized to one, and which is used such that it only has at most one permit available, can serve as a mutual exclusion lock. This is more commonly known as a *binary semaphore*, because it only has two states: one permit available, or zero permits available. When used in this way, the binary semaphore has the property (unlike many `Lock` implementations), that the "lock" can be released by a thread other than the owner (as semaphores have no notion of ownership). This can be useful in some specialized contexts, such as deadlock recovery.

The constructor for this class optionally accepts a *fairness* parameter. When set false, this class makes no guarantees about the order in which threads acquire permits. In particular, *barging* is permitted, that is, a thread invoking `acquire()` can be allocated a permit ahead of a thread that has been waiting - logically the new thread places itself at the head of the queue of waiting threads. When fairness is set true, the semaphore guarantees that threads invoking any of the `acquire` methods are selected to obtain permits in the order in which their invocation of those methods was processed (first-in-first-out; FIFO). Note that FIFO ordering necessarily applies to specific internal points of execution within these methods. So, it is possible for one thread to invoke `acquire` before another, but reach the ordering point after the other, and similarly upon return from the method. Also note that the untimed `tryAcquire` methods do not honor the fairness setting, but will take any permits that are available.

Generally, semaphores used to control resource access should be initialized as fair, to ensure that no thread is starved out from accessing a resource. When using semaphores for other kinds of synchronization control, the throughput advantages of non-fair ordering often outweigh fairness considerations.

This class also provides convenience methods to `acquire` and `release` multiple permits at a time. Beware of the increased risk of indefinite postponement when these methods are used without fairness set true.

Since:

1.5

See Also:

Serialized Form

Constructor Summary

Semaphore(int permits) Creates a Semaphore with the given number of permits and nonfair fairness setting.
Semaphore(int permits, boolean fair) Creates a Semaphore with the given number of permits and the given fairness setting.

Method Summary

void	**acquire**() Acquires a permit from this semaphore, blocking until one is available, or the thread is interrupted.
void	**acquire**(int permits) Acquires the given number of permits from this semaphore, blocking until all are available, or the thread is interrupted.
void	**acquireUninterruptibly**() Acquires a permit from this semaphore, blocking until one is available.
void	**acquireUninterruptibly**(int permits) Acquires the given number of permits from this semaphore, blocking until all are available.
int	**availablePermits**() Returns the current number of permits available in this semaphore.
int	**drainPermits**() Acquire and return all permits that are immediately available.
protected Collection<Thread>	**getQueuedThreads**() Returns a collection containing threads that may be waiting to acquire.
int	**getQueueLength**() Returns an estimate of the number of threads waiting to acquire.
boolean	**hasQueuedThreads**() Queries whether any threads are waiting to acquire.
boolean	**isFair**() Returns true if this semaphore has fairness set true.
protected void	**reducePermits**(int reduction) Shrinks the number of available permits by the indicated reduction.
void	**release**() Releases a permit, returning it to the semaphore.
void	**release**(int permits) Releases the given number of permits, returning them to the semaphore.
String	**toString**() Returns a string identifying this semaphore, as well as its state.
boolean	**tryAcquire**() Acquires a permit from this semaphore, only if one is available at the time of invocation.
boolean	**tryAcquire**(int permits)

	Acquires the given number of permits from this semaphore, only if all are available at the time of invocation.
boolean	**tryAcquire**(int permits, long timeout, TimeUnit unit) Acquires the given number of permits from this semaphore, if all become available within the given waiting time and the current thread has not been interrupted.
boolean	**tryAcquire**(long timeout, TimeUnit unit) Acquires a permit from this semaphore, if one becomes available within the given waiting time and the current thread has not been interrupted.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

Semaphore

```
public Semaphore(int permits)
```

Creates a Semaphore with the given number of permits and nonfair fairness setting.

Parameters:

permits - the initial number of permits available. This value may be negative, in which case releases must occur before any acquires will be granted.

Semaphore

```
public Semaphore(int permits,
                 boolean fair)
```

Creates a Semaphore with the given number of permits and the given fairness setting.

Parameters:

permits - the initial number of permits available. This value may be negative, in which case releases must occur before any acquires will be granted.

fair - true if this semaphore will guarantee first-in first-out granting of permits under contention, else false.

Method Detail

acquire

```
public void acquire()
            throws InterruptedException
```

Acquires a permit from this semaphore, blocking until one is available, or the thread is

interrupted.

Acquires a permit, if one is available and returns immediately, reducing the number of available permits by one.

If no permit is available then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of two things happens:

- Some other thread invokes the release() method for this semaphore and the current thread is next to be assigned a permit; or
- Some other thread interrupts the current thread.

If the current thread:

- has its interrupted status set on entry to this method; or
- is interrupted while waiting for a permit,

then InterruptedException is thrown and the current thread's interrupted status is cleared.

Throws:

InterruptedException - if the current thread is interrupted

See Also:

Thread.interrupt()

---

acquireUninterruptibly

```
public void acquireUninterruptibly()
```

Acquires a permit from this semaphore, blocking until one is available.

Acquires a permit, if one is available and returns immediately, reducing the number of available permits by one.

If no permit is available then the current thread becomes disabled for thread scheduling purposes and lies dormant until some other thread invokes the release() method for this semaphore and the current thread is next to be assigned a permit.

If the current thread is interrupted while waiting for a permit then it will continue to wait, but the time at which the thread is assigned a permit may change compared to the time it would have received the permit had no interruption occurred. When the thread does return from this method its interrupt status will be set.

---

tryAcquire

```
public boolean tryAcquire()
```

Acquires a permit from this semaphore, only if one is available at the time of invocation.

Acquires a permit, if one is available and returns immediately, with the value true, reducing the number of available permits by one.

If no permit is available then this method will return immediately with the value false.

Even when this semaphore has been set to use a fair ordering policy, a call to `tryAcquire()` *will* immediately acquire a permit if one is available, whether or not other threads are currently waiting. This "barging" behavior can be useful in certain circumstances, even though it breaks fairness. If you want to honor the fairness setting, then use `tryAcquire(0, TimeUnit.SECONDS)` which is almost equivalent (it also detects interruption).

Returns:

`true` if a permit was acquired and `false` otherwise.

---

tryAcquire

```
public boolean tryAcquire(long timeout,
                          TimeUnit unit)
                   throws InterruptedException
```

Acquires a permit from this semaphore, if one becomes available within the given waiting time and the current thread has not been `interrupted`.

Acquires a permit, if one is available and returns immediately, with the value `true`, reducing the number of available permits by one.

If no permit is available then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of three things happens:

- Some other thread invokes the `release()` method for this semaphore and the current thread is next to be assigned a permit; or
- Some other thread `interrupts` the current thread; or
- The specified waiting time elapses.

If a permit is acquired then the value `true` is returned.

If the current thread:

- has its interrupted status set on entry to this method; or
- is `interrupted` while waiting to acquire a permit,

then `InterruptedException` is thrown and the current thread's interrupted status is cleared.

If the specified waiting time elapses then the value `false` is returned. If the time is less than or equal to zero, the method will not wait at all.

Parameters:

`timeout` - the maximum time to wait for a permit
`unit` - the time unit of the `timeout` argument.

Returns:

`true` if a permit was acquired and `false` if the waiting time elapsed before a permit was acquired.

Throws:

`InterruptedException` - if the current thread is interrupted

See Also:

`Thread.interrupt()`

---

release

```
public void release()
```

Releases a permit, returning it to the semaphore.

Releases a permit, increasing the number of available permits by one. If any threads are trying to acquire a permit, then one is selected and given the permit that was just released. That thread is (re)enabled for thread scheduling purposes.

There is no requirement that a thread that releases a permit must have acquired that permit by calling `acquire()`. Correct usage of a semaphore is established by programming convention in the application.

---

acquire

```
public void acquire(int permits)
            throws InterruptedException
```

Acquires the given number of permits from this semaphore, blocking until all are available, or the thread is `interrupted`.

Acquires the given number of permits, if they are available, and returns immediately, reducing the number of available permits by the given amount.

If insufficient permits are available then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of two things happens:

- Some other thread invokes one of the `release` methods for this semaphore, the current thread is next to be assigned permits and the number of available permits satisfies this request; or
- Some other thread `interrupts` the current thread.

If the current thread:

- has its interrupted status set on entry to this method; or
- is `interrupted` while waiting for a permit,

then `InterruptedException` is thrown and the current thread's interrupted status is cleared. Any permits that were to be assigned to this thread are instead assigned to other threads trying to acquire permits, as if permits had been made available by a call to `release()`.

Parameters:

`permits` - the number of permits to acquire

Throws:

`InterruptedException` - if the current thread is interrupted

`IllegalArgumentException` - if permits less than zero.

See Also:

`Thread.interrupt()`

---

acquireUninterruptibly

```
public void acquireUninterruptibly(int permits)
```

Acquires the given number of permits from this semaphore, blocking until all are available.

Acquires the given number of permits, if they are available, and returns immediately, reducing the number of available permits by the given amount.

If insufficient permits are available then the current thread becomes disabled for thread scheduling purposes and lies dormant until some other thread invokes one of the `release` methods for this semaphore, the current thread is next to be assigned permits and the number of available permits satisfies this request.

If the current thread is `interrupted` while waiting for permits then it will continue to wait and its position in the queue is not affected. When the thread does return from this method its interrupt status will be set.

Parameters:

`permits` - the number of permits to acquire

Throws:

`IllegalArgumentException` - if permits less than zero.

---

tryAcquire

```
public boolean tryAcquire(int permits)
```

Acquires the given number of permits from this semaphore, only if all are available at the time of invocation.

Acquires the given number of permits, if they are available, and returns immediately, with the value `true`, reducing the number of available permits by the given amount.

If insufficient permits are available then this method will return immediately with the value `false` and the number of available permits is unchanged.

Even when this semaphore has been set to use a fair ordering policy, a call to `tryAcquire` *will* immediately acquire a permit if one is available, whether or not other threads are currently waiting. This "barging" behavior can be useful in certain circumstances, even though it breaks fairness. If you want to honor the fairness setting, then use `tryAcquire(permits, 0, TimeUnit.SECONDS)` which is almost equivalent (it also detects interruption).

Parameters:

`permits` - the number of permits to acquire

Returns:

`true` if the permits were acquired and `false` otherwise.

Throws:

`IllegalArgumentException` - if permits less than zero.

---

tryAcquire

```
public boolean tryAcquire(int permits,
                          long timeout,
                          TimeUnit unit)
                   throws InterruptedException
```

Acquires the given number of permits from this semaphore, if all become available within the given waiting time and the current thread has not been `interrupted`.

Acquires the given number of permits, if they are available and returns immediately, with the value `true`, reducing the number of available permits by the given amount.

If insufficient permits are available then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of three things happens:

- Some other thread invokes one of the `release` methods for this semaphore, the current thread is next to be assigned permits and the number of available permits satisfies this request; or
- Some other thread `interrupts` the current thread; or
- The specified waiting time elapses.

If the permits are acquired then the value `true` is returned.

If the current thread:

- has its interrupted status set on entry to this method; or
- is `interrupted` while waiting to acquire the permits,

then `InterruptedException` is thrown and the current thread's interrupted status is cleared. Any permits that were to be assigned to this thread, are instead assigned to other threads trying to acquire permits, as if the permits had been made available by a call to `release()`.

If the specified waiting time elapses then the value `false` is returned. If the time is less than or equal to zero, the method will not wait at all. Any permits that were to be assigned to this thread, are instead assigned to other threads trying to acquire permits, as if the permits had been made available by a call to `release()`.

Parameters:

`permits` - the number of permits to acquire
`timeout` - the maximum time to wait for the permits
`unit` - the time unit of the `timeout` argument.

Returns:

`true` if all permits were acquired and `false` if the waiting time elapsed before all permits were acquired.

Throws:

`InterruptedException` - if the current thread is interrupted
`IllegalArgumentException` - if permits less than zero.

See Also:

`Thread.interrupt()`

---

release

```
public void release(int permits)
```

Releases the given number of permits, returning them to the semaphore.

Releases the given number of permits, increasing the number of available permits by that amount. If any threads are trying to acquire permits, then one is selected and given the permits that were just released. If the number of available permits satisfies that thread's request then that thread is (re)enabled for thread scheduling purposes; otherwise the thread will wait until sufficient permits are available. If there are still permits available after this thread's request has

been satisfied, then those permits are assigned in turn to other threads trying to acquire permits.

There is no requirement that a thread that releases a permit must have acquired that permit by calling `acquire`. Correct usage of a semaphore is established by programming convention in the application.

Parameters:
`permits` - the number of permits to release
Throws:
`IllegalArgumentException` - if permits less than zero.

---

availablePermits

```
public int availablePermits()
```

Returns the current number of permits available in this semaphore.

This method is typically used for debugging and testing purposes.

Returns:
the number of permits available in this semaphore.

---

drainPermits

```
public int drainPermits()
```

Acquire and return all permits that are immediately available.

Returns:
the number of permits

---

reducePermits

```
protected void reducePermits(int reduction)
```

Shrinks the number of available permits by the indicated reduction. This method can be useful in subclasses that use semaphores to track resources that become unavailable. This method differs from `acquire` in that it does not block waiting for permits to become available.

Parameters:
`reduction` - the number of permits to remove
Throws:
`IllegalArgumentException` - if reduction is negative

---

isFair

```
public boolean isFair()
```

Returns true if this semaphore has fairness set true.

Returns:
true if this semaphore has fairness set true.

---

hasQueuedThreads

```
public final boolean hasQueuedThreads()
```

Queries whether any threads are waiting to acquire. Note that because cancellations may occur at any time, a `true` return does not guarantee that any other thread will ever acquire. This method is designed primarily for use in monitoring of the system state.

Returns:
true if there may be other threads waiting to acquire the lock.

---

getQueueLength

```
public final int getQueueLength()
```

Returns an estimate of the number of threads waiting to acquire. The value is only an estimate because the number of threads may change dynamically while this method traverses internal data structures. This method is designed for use in monitoring of the system state, not for synchronization control.

Returns:
the estimated number of threads waiting for this lock

---

getQueuedThreads

```
protected Collection<Thread> getQueuedThreads()
```

Returns a collection containing threads that may be waiting to acquire. Because the actual set of threads may change dynamically while constructing this result, the returned collection is only a best-effort estimate. The elements of the returned collection are in no particular order. This method is designed to facilitate construction of subclasses that provide more extensive monitoring facilities.

Returns:
the collection of threads

---

toString

`public String toString()`

Returns a string identifying this semaphore, as well as its state. The state, in brackets, includes the String "Permits =" followed by the number of permits.

Overrides:

`toString` in class `Object`

Returns:

a string identifying this semaphore, as well as its state

Submit a bug or feature

For further API reference and developer documentation, see Java 2 SDK SE Developer Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2004 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms. Also see the documentation redistribution policy.