Creating and Starting Threads in Java

From the java tutorials

An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this:

- Provide a Runnable object. The Runnable interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the HelloRunnable example:

```
public class HelloRunnable implements Runnable {

     public void run() {
          System.out.println("Hello from a thread!");
     }

     public static void main(String args[]) {
          (new Thread(new HelloRunnable())).start();
     }

}
```

- Subclass Thread. The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run, as in the HelloThread example:

```
public class HelloThread extends Thread {

     public void run() {
          System.out.println("Hello from a thread!");
     }

     public static void main(String args[]) {
          (new HelloThread()).start();
     }

}
```

Notice that both examples invoke Thread.start in order to start the new thread.

Warning: don't do this

```java
// a class to show what to avoid in java concurrent programming
// NoVisibility: Sharing variables without synchronization; authors Brian Goetz and Tim Peierls

public class ouch {

    private static boolean ready = false;     // quit thread when ready
    private static int number = 0;            // print the resulting number when ready

    // a simple class to run an equally simple thread
    private static class RunningThread extends Thread {

        // simply loop until main class decides it is ready
        public void run() {
            while (!ready)
                Thread.yield();               // give processor to whomever wants it
            System.out.printf("number is now %d\n", number);
        }
    }

    // just running a thread, so no arguments
    public static void main(String args[]) {

        (new RunningThread()).start();

        try { Thread.sleep(1); }              // wait a (long) while
        catch (InterruptedException dont_care) { /* for this example, it doesn't matter */ }

        number = 42;                          // the answer to life, the universe, everything...
        ready = true;
        System.out.printf("main has performed its tasks, and is now quitting.\n");
    }
}
```

Semaphores in Java

Straight from the horse's mouth

java.util.concurrent

`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 int 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;
   }
 }
```

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.

Memory consistency effects: Actions in a thread prior to calling a "release" method such as `release()` *happen-before* actions following a successful "acquire" method such as `acquire()` in another thread.

Since:

1.5

Constructors

Constructor and Description
`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*

All Methods Instance Methods Concrete Methods

Modifier and Type	Method and Description
`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`**`()` Acquires and returns 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**.