Semaphores

Initializing

A Semaphore object can best be described as a lock with a counter.

We can create a Semaphore by passing the number of **permits** as an argument. A permit is required whenever a thread wants to access a resource guarded by the Semaphore.

So, this counter determines the maximum number of threads that can gain entry simultaneously.

// Create a Semaphore with two permits to issue.

Semaphore semaphore = new Semaphore(2);

The Java API also allows us to define a semaphore's **fairness** by setting the **fair flag** of the constructor.

The newly initialized Semaphore will then work on a **FIFO** (first in, first out) basis.

The following access will be granted to the thread in the queue that has been blocked for the longest time.

Semaphore fairSemaphore = new Semaphore(2, true);

Note: if you don't explicitly set the fair flag in the constructor, it will be false by default.

Some important methods

Some important methods

Now we have our customized Semaphore, but how can we get permits from it?

- The public void acquire() method will obtain a permit and reduce the number of available permits by one. If there are no permits left, the calling thread will be blocked.

- If a permit is available, the public boolean tryAcquire() method acquires a permit and decrements the counter similar to acquire(). Then, it returns true. Otherwise, no permit is obtained, and false is returned.

The return value is not the only thing that differentiates these two functions.

The astute reader might have already noticed that we haven't said anything about what happens to the current thread if tryAcquire() returns false.

Well, this is because tryAcquire() returns immediately, regardless of the result of the call. Consequently, the calling thread will not try to access the resource again, even if a permit can be acquired. On the other hand, if a thread is blocked, it will remain blocked until a permit is available again.

```
semaphore.acquire();
System.out.println(semaphore.availablePermits()); // = 1
semaphore.acquire();
System.out.println(semaphore.availablePermits()); // = 0
System.out.println(semaphore.tryAcquire()); // false
semaphore.acquire(); //blocked
```

The return value is not the only thing that differentiates these two functions. The astute reader might have already noticed that we haven't said anything about what happens to the current thread if tryAcquire() returns false. Well, this is because tryAcquire() returns immediately, regardless of the result of the call. Consequently, the calling thread will not try to access the resource again, even if a permit can be acquired. On the other hand, if a thread is blocked, it will remain blocked until a permit is available again.

```
semaphore.acquire();
System.out.println(semaphore.availablePermits()); // = 1
semaphore.acquire();
System.out.println(semaphore.availablePermits()); // = 0
System.out.println(semaphore.tryAcquire()); // false
semaphore.acquire(); //blocked
```

The release() method will give us the answer. A thread calling the release() function willingly gives up its access to the shared resource while incrementing the counter of the Semaphore by one. As we said, if a thread happens to be waiting when a permit is released, it will be unblocked and gain access.

```
semaphore.release();
// A blocked thread will get access after this point
```

Who decides which thread will gain access next? The Semaphore? The programmer? The truth lies somewhere in the middle. This is where our **fair flag** comes into play. By default, if multiple threads are waiting, it is hard to predict which thread gains access next. But if you have the fair flag set, the threads will unblock in the order they started waiting. This is a simple way to prevent **thread starvation**, in which a thread is always waiting for other threads because of randomness.

The parking lot

Let's conclude our acquaintance with Semaphores by elaborating on our previous analogy! Imagine that many shoppers are flooding the mall's parking lot (it's Black Friday, after all). They are all trying to park, but only two spots are left. This little demo program illustrates the situation using the Semaphore class.

```
public class ParkingLot {
    public static void main(String[] args) {
        Semaphore sem = new Semaphore(2);
        for (int i = 0; i < 50; ++i) {
            Car car = new Car("Car #" + i, sem, 3000L);
            car.goShopping();
        }
    }
}
class Car extends Thread {
    private final Semaphore semaphore;
    private final long timeout; // ms

    public Car(String name, Semaphore semaphore, long timeout) {
        super(name);
        this.semaphore = semaphore;
        this.timeout = timeout;
    }

    public void goShopping() {
        start();
    }

    @Override
    public void run() {
        try {
            if (!semaphore.tryAcquire()) {
                System.out.println(Thread.currentThread().getName() + " waits for
parking");
                semaphore.acquire();
            }
            System.out.println(Thread.currentThread().getName() + " parked");
            Thread.sleep(timeout); // shopping
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + ": shopping was
interrupted");
            e.printStackTrace();
        } finally {
            System.out.println(Thread.currentThread().getName() + " left");
            semaphore.release();
```

```
      }
    }
}
```

As you can see from the standard output, all the running threads are trying to acquire a permit (parking space) from the Semaphore, but only two will be allowed at a time.

```
...
Car #1 parked
Car #5 parked
Car #2 waits for parking
Car #4 waits for parking
Car #3 waits for parking
Car #6 waits for parking
Car #1 left
Car #5 left
Car #2 parked
Car #3 parked
Car #3 left
Car #2 left
Car #4 parked
Car #6 parked
...
```

The sleep() method is not an essential part of this code — its only purpose is to "slow down" the threads, so we can more easily follow what's happening. You can try it by running the code without the sleep() call — the program's behavior will not change.