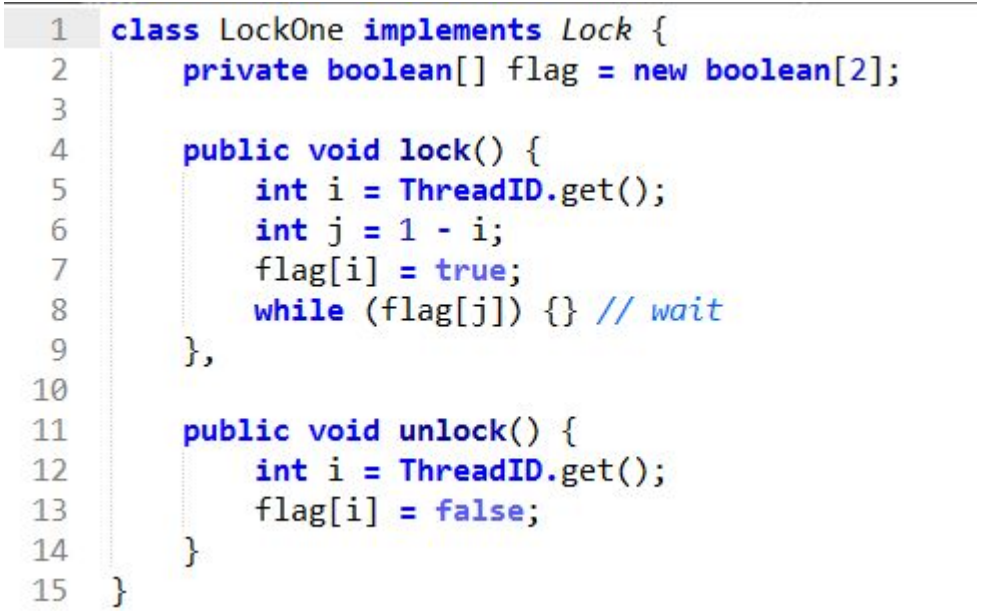
How to implement mutex?

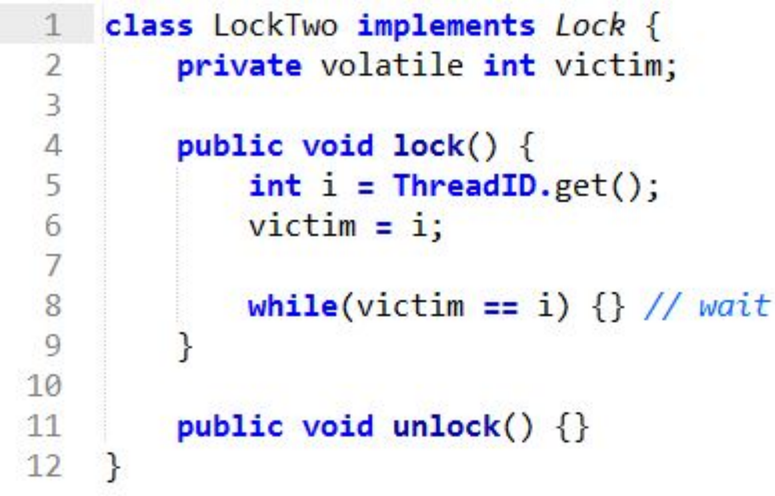


```java
class LockOne implements Lock {
    private boolean[] flag = new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {} // wait
    },

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

How to implement mutex? Option 2



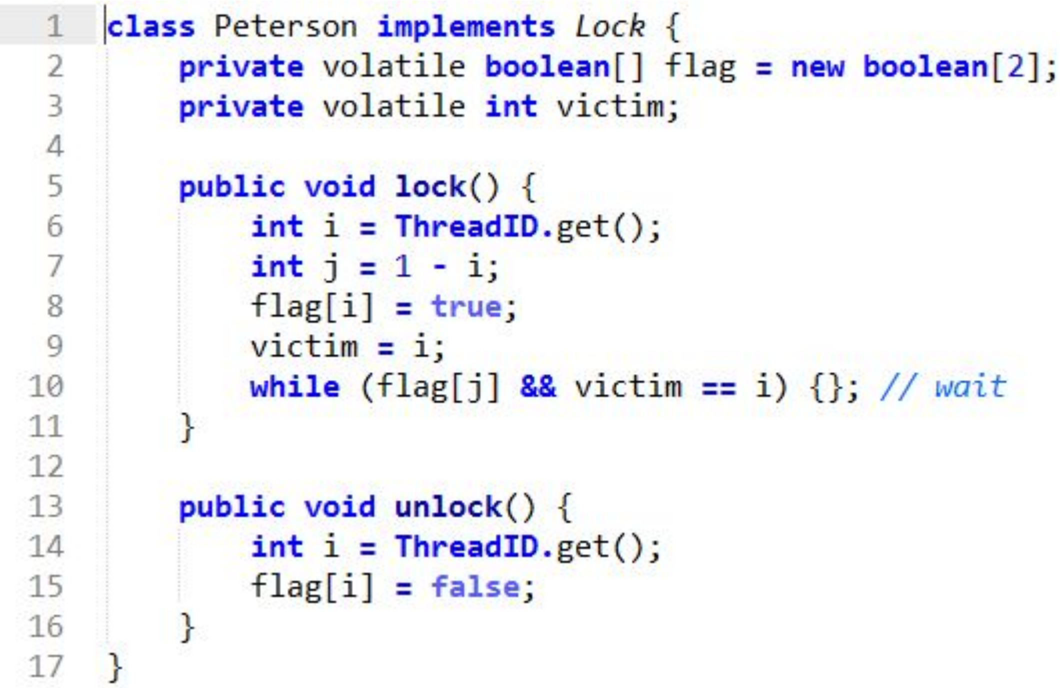
```java
class LockTwo implements Lock {
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;

        while(victim == i) {} // wait
    }

    public void unlock() {}
}
```

Working version



```java
class Peterson implements Lock {
    private volatile boolean[] flag = new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {}; // wait
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

Synchronization primitives

Non-recursive - when re-capturing by the same thread, call **deadlock**

Recursive - allow you to capture yourself by the same thread multiple times

Recursive primitives

```
Lock
  Lock
    Lock
    Unlock
  Unlock
Unlock
```

Method1: Lock

Method2: Lock

Method3: Lock

Method4: Unlock

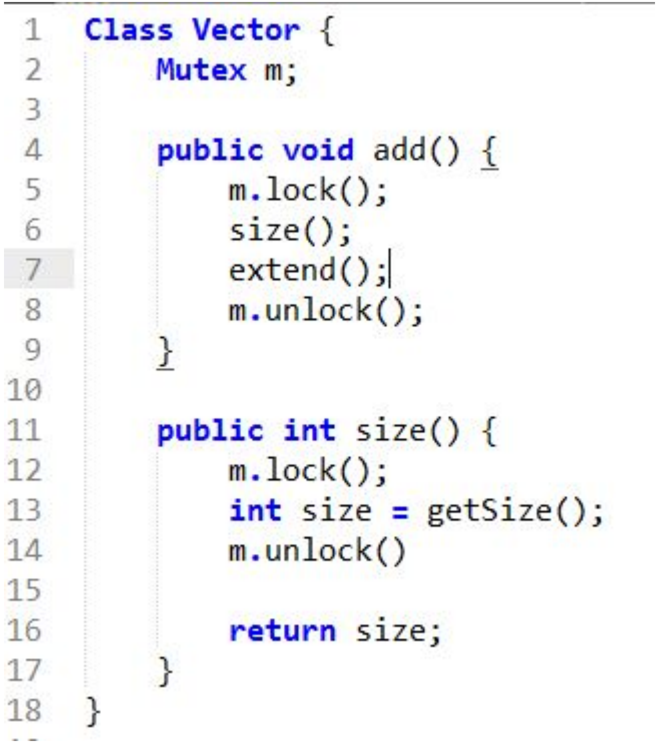
Method5: Lock

Method6: Unlock

Method7: Unlock

Method8: Unlock

Example 1



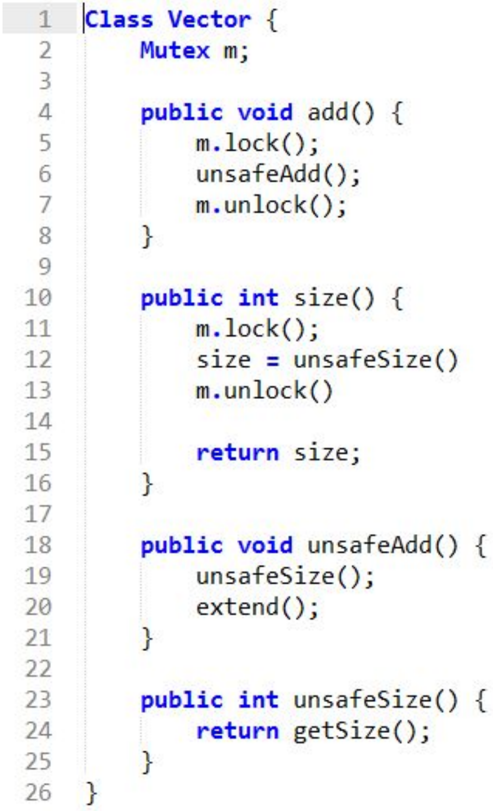
```
Class Vector {
    Mutex m;

    public void add() {
        m.lock();
        size();
        extend();
        m.unlock();
    }

    public int size() {
        m.lock();
        int size = getSize();
        m.unlock()

        return size;
    }
}
```

Example 2



```
Class Vector {
    Mutex m;

    public void add() {
        m.lock();
        unsafeAdd();
        m.unlock();
    }

    public int size() {
        m.lock();
        size = unsafeSize()
        m.unlock()

        return size;
    }

    public void unsafeAdd() {
        unsafeSize();
        extend();
    }

    public int unsafeSize() {
        return getSize();
    }
}
```

Types of mutexes

- Timed mutex
- Shared mutex
- Spin mutex
- Futex

Timed mutex

Tries to capture the mutex within the specified time

Shared mutex

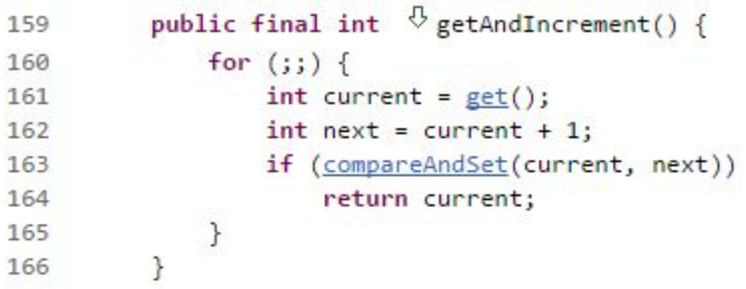
Allows you to **lock** read-only, write-only, or mixed. Allows you to collapse the **lock-on** separate operations into a single lock.

w r r r w r r w r r r = w r w r w r

Spin mutex

Active waiting

java.util.concurrent.atomic.AtomicInteger



```
public final int getAndIncrement() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return current;
    }
}
```

CAS - operations

CAS - compare and set, compare and swap

bool compare_and_set(int *source, int oldValue, int newValue)

- Returns whether the value was set successfully
- Atomic at the processor level (i486+): **cmpxchg**

Examples:

a = 5; current_value = a;

compare_and_set(&a, current_value, 6)

SPIN lock - Active waiting

Use a lot of CPU time

When to use?

SPIN lock - Active waiting

Spinlock can be better when you plan to hold the lock for an extremely short interval (for example to do nothing but increment a counter), and contention is expected to be rare.

Benefits:

- On unlock, there is no need to check if other threads may be waiting for the lock and waking them up. Unlocking is simply a single atomic write instruction.
- Failure to immediately obtain the lock does not put your thread to sleep, so it may be able to obtain the lock with much lower latency as soon a it does become available.