Locks

java.util.concurrent.locks

- A lock (*fr.* verrou) is a more flexible version of a synchronized statement
- The synchronized statement in

```
Object someObject = new Object();

void m() {
    synchronized(someObject) {
        ...
    }
}
```

Only one thread can enter. Other threads have to wait until the thread finishes the synchronized statement.

is more or less equivalent to

```
private final ReentrantLock lock = new ReentrantLock();

public void m() {
    lock.lock();
    try {
        ...
    } finally {
        lock.unlock();
    }
}
```

Only one thread can lock(). Other threads have to wait until the thread calls unlock()

finally is used here to ensure that unlock() is always called

Locks vs Synchronized

- Locks are more general than `synchronized` statements
- For example, you can call lock() and unlock() in different places:

```
final ReentrantLock lock = new ReentrantLock();

private void lockMyList() {
    lock.lock(); System.out.println("lock");
}

private void unlockMyList() {
    lock.unlock(); System.out.println("unlock");
}

void add(int value) {
    lockMyList();
    ...
    unlockMyList();
}
```

```
void remove(int value) {
    lockMyList();
    ...
    unlockMyList();
}
```

- This makes the structure of your program more readable sometimes.
  - Of course, this is dangerous. If you forget to call unlockMyList(), the lock is never released!

Locks vs Synchronized (2)

- Another thing you cannot do with `synchronized`:

  Test whether the lock is already locked by another thread

- Example:

```
if (lock.tryLock()) {
    try {
        ...
    } finally {
        lock.unlock();
    }
} else {
    // don't wait for the lock.
    // do something else
}
```

- The method tryLock()
  - acquires the lock and returns true if it is open
  - returns false if the lock is already locked by another thread

Again our photo application...

```
// Code for T1
while(true) {
    Picture currentPicture = takePhoto();
    synchronized(someObject) {
        while(picture!=null) {
            try {
                someObject.wait();
            }
            catch(InterruptedException e) {  throw new RuntimeException("...", e); }
        }
        picture = currentPicture;
        someObject.notify();
    }
}

// Code for T2:
while(true) {
    Picture currentPicture;
    synchronized(someObject) {
        while(picture==null) {
            try {
                someObject.wait();
            }
            catch(InterruptedException e) { throw new RuntimeException("...", e); }
        }
        currentPicture=picture;
        picture=null;
        someObject.notify();
    }

    CompressedPicture p=compress(currentPicture);
    p.writeToFile();
}
```

Our program is not very easy to understand because we are using someObject for three things:

1. Make sure that the picture variable cannot be accessed by two threads at the same time
2. Wait/notify when there is no picture (picture==null)
3. Wait/notify when there is a picture (picture!=null)

Improved version with locks (only the code for thread T1)

```
Picture picture;
final ReentrantLock lock = new ReentrantLock();
final Condition noPicture  = lock.newCondition();
final Condition havePicture = lock.newCondition();


 // New code for T1
 while(true) {
     Picture currentPicture = takePhoto();
     lock.lock();
     try {
         while(picture!=null) {
             try {
                 noPicture.await();
             }
             catch(InterruptedException e) {
                throw ...
             }
         }
         picture = currentPicture;
         havePicture.signal();
     }
     finally {
         lock.unlock();
     }
 }
```

Old version without locks

```
Picture picture;
Object someObject = new Object();

// Old code for T1
while(true) {
    Picture currentPicture = takePhoto();
    synchronized(someObject) {
        while(picture!=null) {
            try {
                someObject.wait();
            }
            catch(InterruptedException e) {
                throw ...
            }
        }
        picture = currentPicture;
        someObject.notify();
    }
}
```

Conditions

- Conditions work like wait()/notify():
  - await(), signal(), signalAll() = wait(), notify(), notifyAll()
  - you can also specify a timeout with await(...) (like wait(...))
  - the thread must own the lock before it can use the condition

- Advantage of conditions over wait()/notify(): A lock can have more than one condition (see example on the previous slide).
  - Makes the program easier to understand
  - signal() only wakes up those threads that are waiting for exactly that condition.