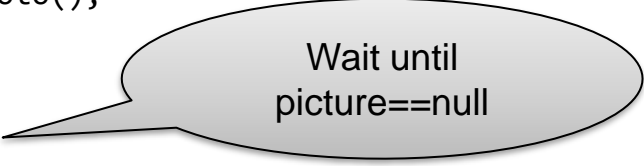


Visibility, deadlocks, and more

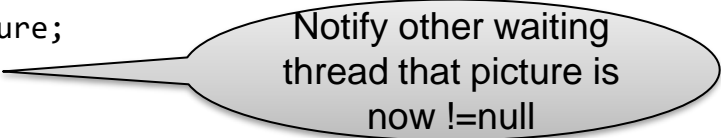
wait() and notify()

```
// 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();  
    }  
}
```



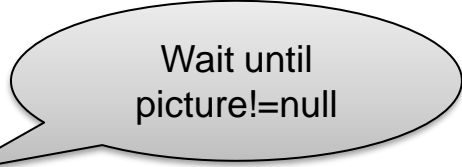
Wait until
picture==null



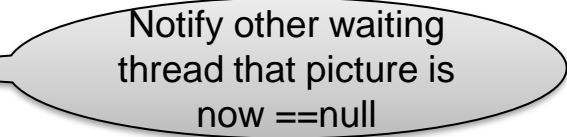
Notify other waiting
thread that picture is
now !=null

```
// 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();  
    }  
}
```



Wait until
picture!=null



Notify other waiting
thread that picture is
now ==null

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

Waiting with timeout

- Sometimes, you want to limit the time to wait (BoundedBuffer exercise on ingenious):

```
someObject.wait(2000);    // wait maximum 2000ms
```

- The thread stops waiting if:
 - another thread calls notify() or notifyAll()
 - the time is over
 - the waiting is interrupted (InterruptedException)
- Don't forget to test after waiting if the condition you were waiting for is satisfied.

Visibility

- The `synchronized` statement also does something else: It guarantees the visibility of data modifications to threads
- Incorrect example:

Thread 1

```
someObject.b=true;
...
while(someObject.b) {
    ...
}
```

Thread 2

```
someObject.b=false;
```

- Will thread 1 terminate?
- We don't know! In Java, it is not guaranteed that thread 1 sees modifications made by thread 2 unless thread 1 and thread 2 synchronize (for example with a synchronized statement)

Visibility (2)

- Using a synchronized statement is one way to ensure the visibility of modifications
- It is also possible to declare a class member as *volatile*:

```
class SomeClass {  
    volatile boolean b;  
}
```

- When a class member is volatile, Java guarantees that a thread reading the variable

Example: `while(someObject.b) { ...`

will see all previous modifications by other threads

Example: `someObject.b=false;`

Deadlocks

- A thread can own more than one monitor. But be careful with deadlocks!

```
Object obj1=new Object();  
Object obj2=new Object();  
Thread t1=new Thread(() -> {
```

```
    synchronized(obj1) {
```

```
        try {
```

```
            Thread.sleep(1000);
```

```
        }
```

```
        catch(InterruptedException e) {
```

```
        }
```

```
        synchronized(obj2) {
```

```
        }
```

```
    }
```

```
};
```

```
Thread t2=new Thread(() -> {
```

```
    synchronized(obj2) {
```

```
        synchronized(obj1) {
```

```
        }
```

```
    }
```

```
};
```

```
t1.start(); t2.start();
```

```
t1.join(); t2.join();
```

Thread 1 gets monitor of obj1

Thread 1 waits for monitor of obj2

Thread 2 gets monitor of obj2

Thread 2 waits for monitor of obj1

Thread 1 and thread 2 block
each other. They can't finish.
Deadlock!



- How can a computer execute multiple threads at the same time? Why don't threads see modifications made by other threads?
 - LINFO1252: Systèmes informatiques
 - LINGI2241: Computer architecture and performance
 - LINGI2355: Multicore programming
- How do you design algorithms and programs with threads? How can you prove that a program with multiple threads works correctly?
 - LINFO1104: Paradigmes de programmation et concurrence
 - LINGI2143: Concurrent systems : models and analysis
 - LSINF2345: Languages and algorithms for distributed applications