



T-106.420  
Concurrent Programming  
Java Concurrency Features



# Concurrency in Java: Threads

- Four steps to get threads in Java:
- Define a *new thread class* by extending class Thread:

```
class MyThread extends Thread() {  
    public void run() {... } // thread code here  
}
```

- Put the *thread code* in: `public void run() {... }`
- Create one `MyThread` *instance*, ie. an *object* with reference `s` :

```
MyThread s = new MyThread();
```

- *Start the execution* of thread `s` :

```
s.start(); //this will call the run() method in  
object s
```



# Concurrency in Java: interface Runnable

- Java doesn't allow extensions from several classes, i.e. multiple inheritance. A way is needed to create threads from any class.
- Any existing class to can act as a thread if implements a special interface `Runnable` including a method `run()`

- Declaring of `MyClass` as runnable:

```
class MyClass implements Runnable() {  
    public void run() {... }  
}
```

- Instantiation of a runnable object `s`:

```
Runnable s = new MyClass();
```

- Instantiation `s` as a thread and starting it:

```
new Thread(s).start();
```



# A monitor in Java: A class with one or more synchronized methods

- Any class can act as monitor for threads
  - Shared vs. non shared state:
    - Shared: Private, permanent variables of the shared object
    - Non shared: Parameters and local variables of individual method activations
  - Mutual exclusion of monitor procedures:
    - Synchronized methods
  - Also only parts of the monitor methods can be mutually excluded:
    - Synchronized blocks within methods



# Synchronization in Java monitors: Only one delay queue per Object

- One implicit condition variable (delay queue) per object, so parameter list `()` is empty:
  - `wait()`: release the (object) lock, wait in the (object) queue, release the processor
  - `notify()`: like signal with SC semantics
  - `notifyAll()`: like `signal_all`  
these may be used only inside synchronized methods or blocks
- Implementation: only one waiting queue per object needed for both mutual exclusion and synchronization.



## Consider the following Java solution for Readers and Writers:

```
class ReadersWriters {
    int nr = 0;
    private synchronized void startRead() {
        nr++;
    }
    private synchronized void endRead() {
        nr--;
        if (nr==0) notify();//awaken waiting Writers
    }
    public void read() {
        startRead();
        ... // embed read operation here!!
        endRead();
    }
    public synchronized void write() {
        while (nr>0)
            try { wait(); }
                catch (InterruptedException ex)
                    {return;}
        ... // embed write operation here!!
        notify();
    }
}
```

a) Does it exclude concurrent access during writing?

b) Explain the roles of the three methods:

```
public void read(),
private synchronized void
startRead(),
private synchronized void endRead()
```

c) Does it provide concurrent access to the readers?

d) What is the purpose of the line:

```
catch (InterruptedException ex)
    {return;}?
```

e) Is the solution fair? Explain.

f) Why the is the `while (nr>0) {...}` loop needed in the `write()` ?

g) Modify the solution so that it gives preference to the writers.



## A Java solution for Readers and Writers:

- a) Yes, because writing is done inside a synchronized procedure.
- b) `public void read()` is used to provide one interface to the clients. `private synchronized void startRead()`, `private synchronized void endRead()` are used for the synchronization needed for reading.
- c) Yes, because the procedure `public void read()` is public and not synchronized, so that the actual reading part ... in it is not mutually excluded
- d) It enables to catch an exception to escape from the wait without writing in the case of some possible exception, like a fault of another process, deadlock etc. This is a standard Java programming pattern used with the wait primitive.
- e) No. Writers may have to wait indefinitely if there is always at least one reader active ie.  $nr > 0$  holds.
- f) Because of the Signal-and-Continue (SC) semantics in Java, the writer can not be sure that no reader could not “sneak into the monitor ” between the notify and its activation from wait.
- g) (ref. to 4 following slides)



## A Java solution for Writer preference:

```
class ReadersWriters {
  int nr = 0, iw = 0,                    #1.
  nw = 0;
  private synchronized void startRead() {
    while (iw>0)                         #1.
      try { wait(); }
      catch (InterruptedException ex) {return;}
    nr++;
  }
  private synchronized void endRead() {
    ...
  }
  private synchronized void startWrite() {
    iw++;                                  #1.
    while (nr>0 or nw >0)
      try { wait(); }
      catch (InterruptedException ex) {return;}
    ...
    iw--;                                  #1.
  }
  private synchronized void endWrite() {
    ...
  }
  public void write(){
  ... }}
```

Changes:

# 1. To exclude reading when the number of *interested writers*, *iw* is positive, i.e. ( $iw > 0$ ), `start_read()` procedure will put the readers to wait in order to give preference to interested writers.



## A Java solution for Writer preference:

```
class ReadersWriters {
  int nr = 0, iw = 0,
  nw = 0; #2.
  private synchronized void startRead() {
  ...
  }
  private synchronized void endRead() {
  ...
  }
  private synchronized void startWrite() { #3.
  ...
  while (nr>0 or nw>0) #2.
    try { wait(); }
    catch (InterruptedException ex) {return;}
  nw++; #2.
  ...
  }
  private synchronized void endWrite() { #3.
  nw--; #2.
  notifyAll(); //awaken all possible waiters!!
  }
  public void write() { #3.
  startWrite();
  ... // embed the real write operation here!!
  endWrite()
  } }
}
```

Changes:

**# 2.** In order to be able to schedule the waiting writers against the readers, the writers have to wait inside the monitor for the condition  $(nr>0 \text{ or } nw>0)$ . To exclude writers from each other an integer variable `nw` is needed to indicate that a write is ongoing.

**# 3.** Writing is split to three procedures like reading: `startWrite()`, `endWrite()` and `write()` to enable interested writers to enlist themselves during writing. This is necessary to ensure strict preference to the writers independent of Java cond queue implementation.



## A Java solution for Writer preference:

```
class ReadersWriters {
    int nr = 0, iw = 0,
        nw = 0;

    private synchronized void startRead() {
        ...
    }
    private synchronized void endRead() {
        nr--;
        if (nr==0) notifyAll(); //wake all!! #4.
    }
    public void read() {
        ...
    }
    private synchronized void startWrite() {
        ...
    }
    private synchronized void endWrite() {
        nw--;
        notifyAll(); //wake all !! #4.
    }
    public void write() {
        ...
    }
}
```

Changes:

**# 4.** Because of Java's restriction to have only one condition variable per synchronized object, both readers and writers have to wait behind it for their specific signaling condition in some random order. In order to enable all the waiting processes to evaluate their eligibility to proceed after signal they all have to be notified with `notifyAll()`;



```
class ReadersWriters {
int nr = 0, iw = 0,
nw = 0;
private synchronized void startRead() {
while (iw>0)
try { wait(); } catch (InterruptedException ex){return;}
nr++;
}
private synchronized void endRead() {
nr--;
if (nr==0) notifyAll(); //awaken possible Writers
}
public void read() {
startRead();
... // embed read operation here!!
endRead();
}
private synchronized void startWrite() {
iw++;
while (nr>0 or nw>0)
try { wait(); }catch (InterruptedException ex){return;}
iw--; nw++;
}
private synchronized void endWrite() {
nw--;
notifyAll(); //awaken all possible waiters!!
}
public void write(){
startWrite();
... // embed write operation here!!
endWrite()
}
}}
```

11/13

## A Java solution for Readers and Writers with Writer preference: The Whole Story

Exercise: How to  
provide fairness?

11