Technology Session III

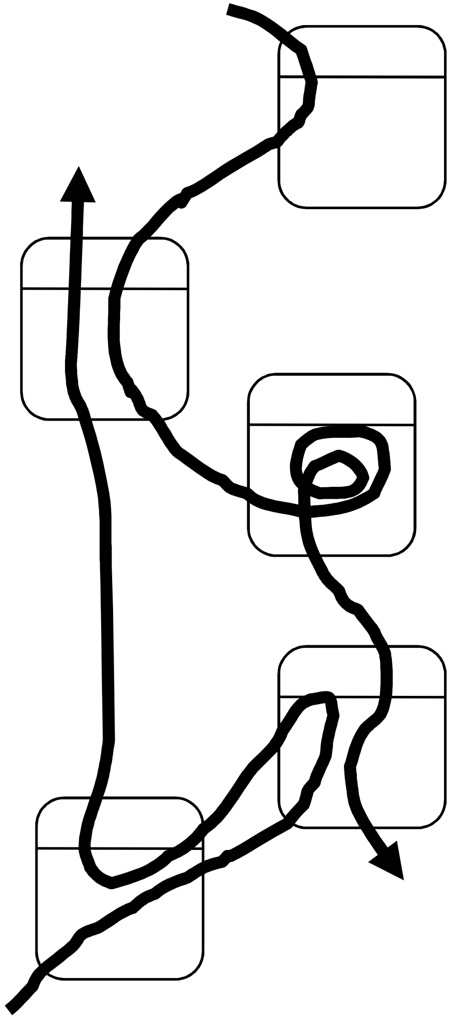
- Concurrency -

Chair of Programming Methodology

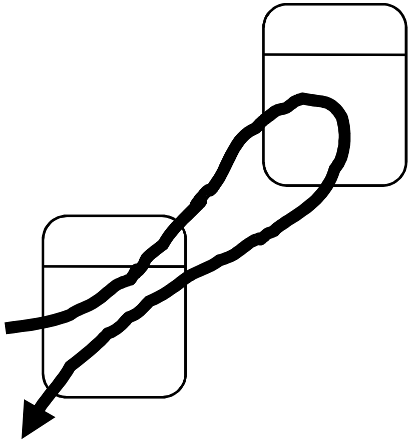
Material based on Prof. Peter Müller's
Konzepte objektorientierter Programmierung
course

Threads

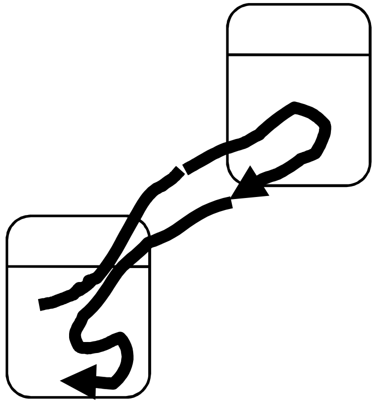
- Execution threads are **sequences of atomic actions** during a program execution
- Concurrent programs can have **more than one thread**
- Execution of threads can be **parallel** (on several processors) or **virtually parallel** (on one processor)
- A **scheduler** maps threads to processors



Concurrency in OO-Programs



- Passive objects
  - Threads **pass through different objects** (by method invocations)
  - **Several threads** executed **on one object** possible



- Active objects
  - **Each object has** its own **thread**
  - Upon method invocation, the thread of the target object serves the request
  - **At most one thread** executed on one object

Threads and Passive Objects

- Threads have to be created, started, synchronized, and controlled
- Threads are represented by special objects
- Method "start" starts new thread and returns immediately

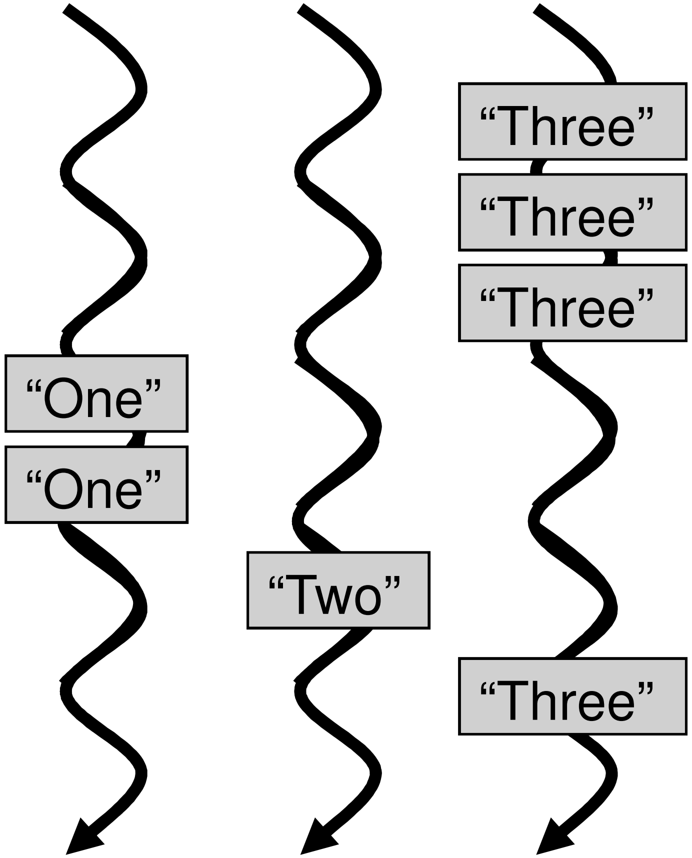
```
interface Runnable {
  void run( );
}
```

```
class Thread
        implements  Runnable {
  Thread( Runnable target )   { … }
  void run( )                 { … }
  native void start( );
  void interrupt( )           { … }
  …
}
```

Example

```
class Printer implements Runnable {
  String val;
  Printer( String s ) { val = s; }
  void run( ) {
    while( true )
      System.out.println( val );
  }
}
```

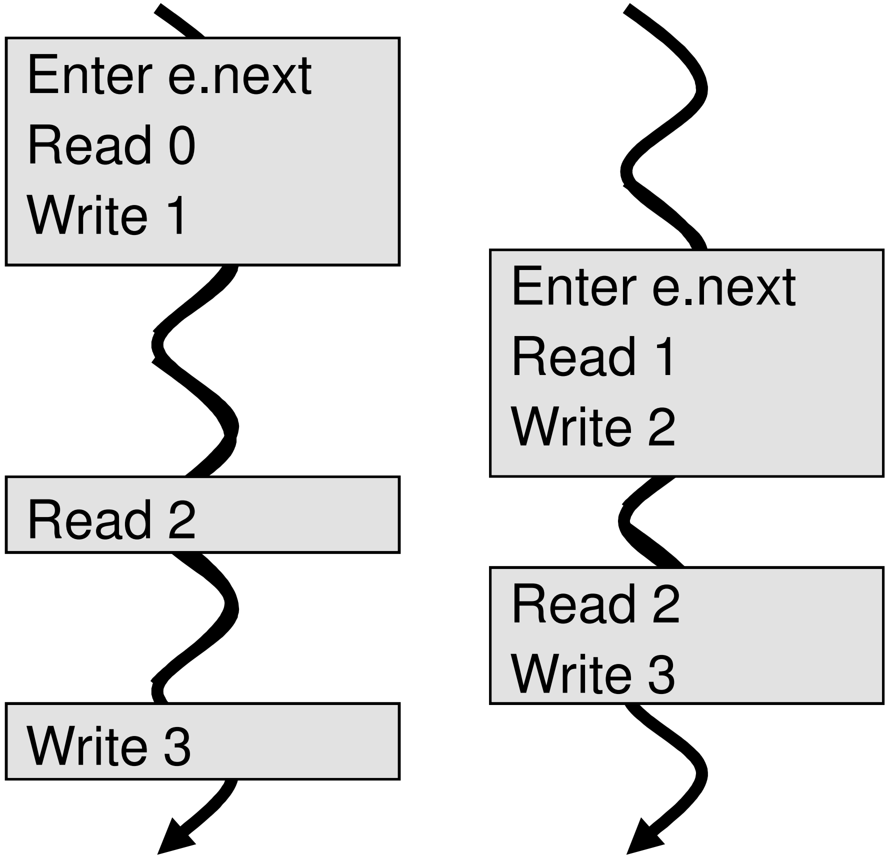
```
new Thread( new Printer( “One” ) ).start( );
new Thread( new Printer( “Two” ) ).start( );
new Thread( new Printer( “Three” ) ).start( );
```



Data Races

- Access to **common resources** (e.g., variables) can lead to unwanted behavior
- Execution is divided into **critical** and non-critical **sections**
- Execution of **critical sections** should be **mutually exclusive**

```
class Even {
  private int x;
  void next( ) { x++; x++; }
}
```



Enter e.next
Read 0
Write 1

Read 2

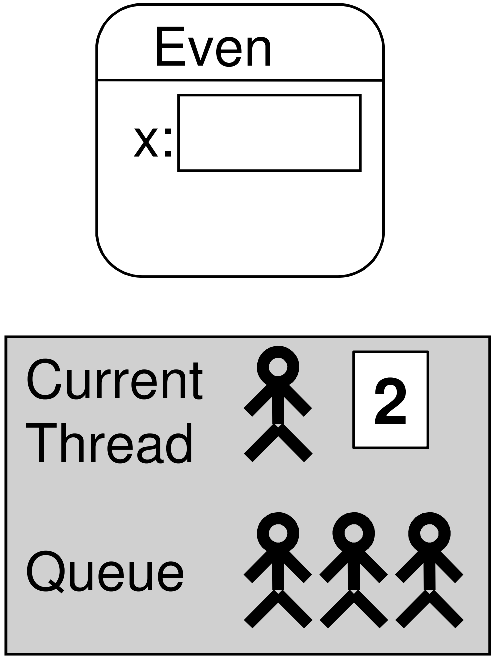
Write 3

Enter e.next
Read 1
Write 2

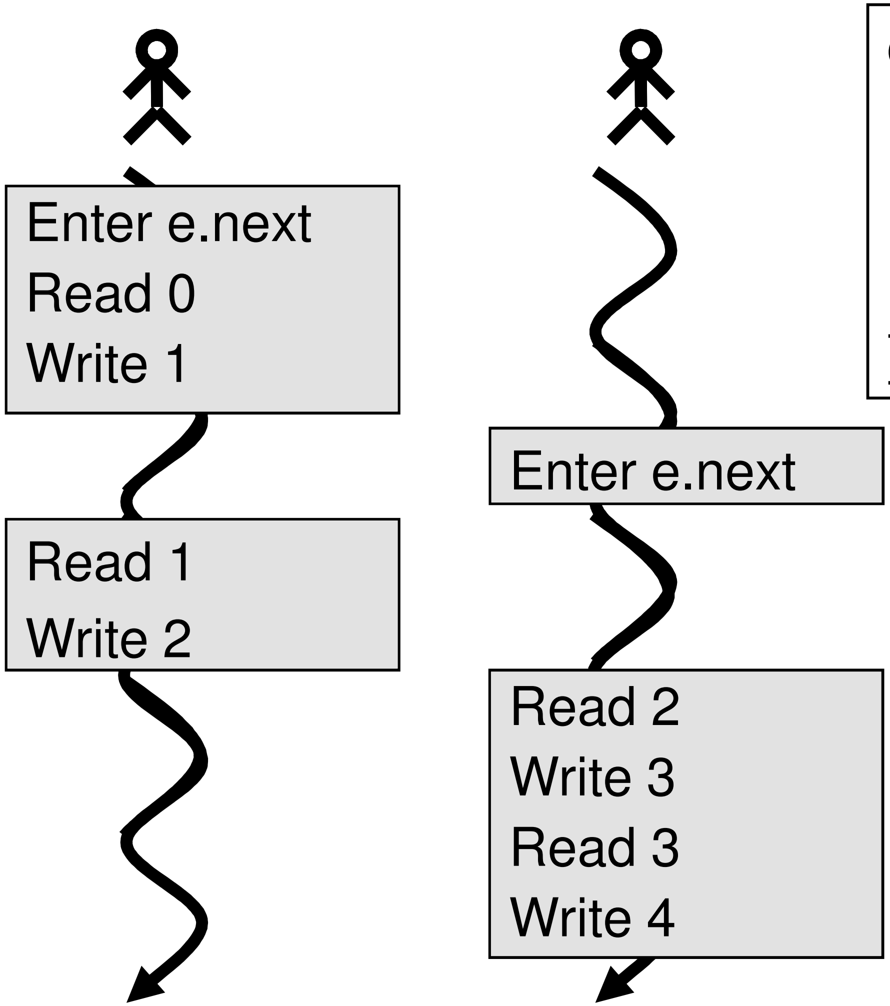
Read 2
Write 3

Object-Oriented Monitors

- Each object has a monitor
- Execution of **synchronized** methods requires lock of monitor
    - Lock is obtained upon invocation
    - Lock is released upon termination
    - Other threads have to wait
- Monitor keeps track of
    - Thread that has locked the monitor
    - Number of locks of this thread
    - Queue of blocked threads



Preventing Data Races



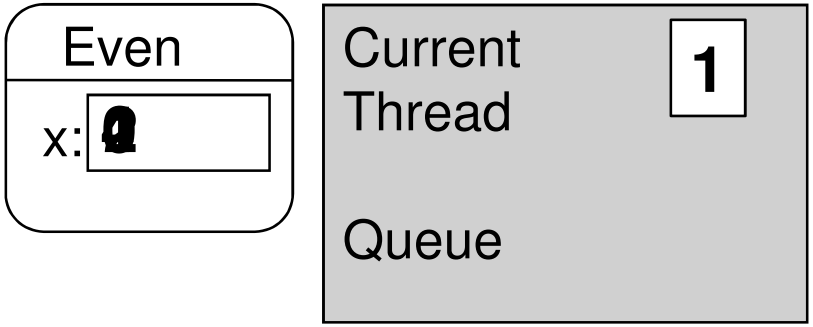
Enter e.next
Read 0
Write 1

Read 1
Write 2

Enter e.next

Read 2
Write 3
Read 3
Write 4

```
class Even {
  private int x;
  synchronized void next( )
      { x++; x++; }
}
```



Even

x: [illegible]

Current Thread **1**

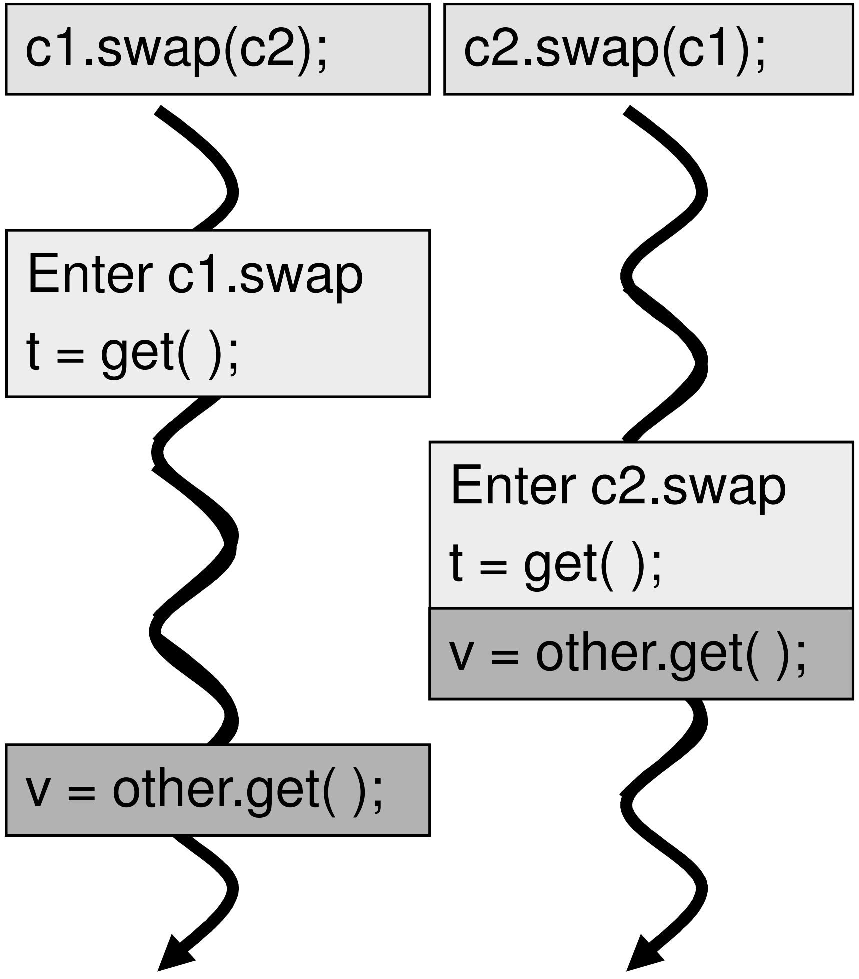
Queue

Safety and Liveness

- Safety
    - **"Nothing bad ever happens"**
    - To perform method actions only when in consistent states
    - Achieved by mutual exclusion
- Liveness
    - **"Something eventually happens"**
    - Every called method should eventually execute
    - Avoiding deadlocks
    - Avoiding unfair scheduling (not guaranteed in Java)

Deadlock Example

```
class Cell {
  private long value;
  synchronized long get( )
    { return value; }
  synchronized void set( long v )
    { value = v; }
  synchronized void
            swap( Cell other ) {
    long t = get( );
    long v = other.get( );
    set( v );
    other.set( t );
  }
}
```



c1.swap(c2);

c2.swap(c1);

Enter c1.swap

t = get( );

Enter c2.swap

t = get( );

v = other.get( );

v = other.get( );