Java-Threads

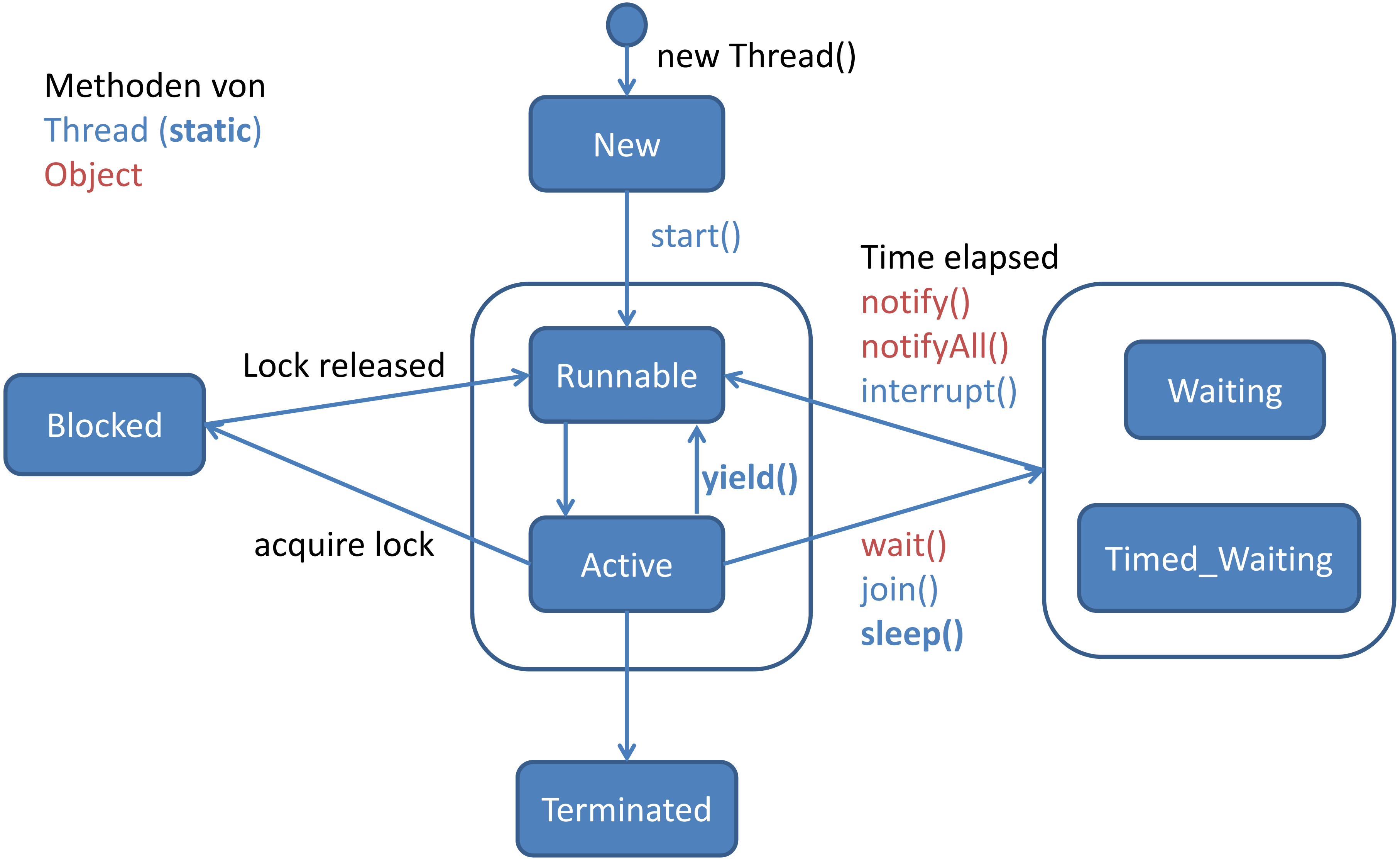
Thread-Parallelization in Java

State of Java Threads

- A Thread is in either on of the following states
  - New
  - Runnable
  - Active
  - Blocked
  - Waiting (Timed-Waiting)
  - Terminated

State of Java Threads

Methoden von
Thread (**static**)
Object



Starting a Thread in Java

- **A thread has to know in which line of code it starts**
- **Idea**
  - The new thread calls a method
  - The thread is destroyed after the method has ended
- **Problem**
  - A function pointer would be good, but since Java has no function pointers, there is another method:
  - Calling the native Thread-class with an own object, implementing the Runnable-Interface

Starting a Thread in Java - Runnables

```
public class MyRunnable implements Runnable
{
   public void run()
   {
     // do something useful
   }
}

[...]

Thread t = new Thread(new MyRunnable());
t.start();

[...]
```

Starting a Thread in Java – Live Demo

- **ThreadBasics - startingThreads**

Ending a Thread in Java

- **A thread will destroy itself when the method, that it was executing, is over**

- **Question**
  - Is there a way to wait unless a thread finishes?

- **Answer**
  - Yes!

```
Thread t = new Thread (new MyRunnable());
t.start();
// do something useful
[...]
t.join(); // wait for thread
```

Ending a Thread in Java – Live Demo

- ThreadBasics - endingThreads

Motivation for synchronizing Threads

- **To avoid race conditions, it’s sometimes necessary to synchronize threads**
  - Synchronization means to actively effect the order of the threads execution

- **There are several methods to realize a synchronization**
  - Atomic operations / atomic data types
  - Mutex locks
  - Barriers
  - …

Race Conditions – Live Demo

- **ThreadSynchronisation1 - withoutSynchronisation**

Atomic operation / Atomic data type

- **An atomic operation is a non-interruptible operations**
  - No other thread or process can perform an operation, while the atomic operation is executed

- **An atomic data type is a data type which operations are atomic**
  - For example **`AtomicInteger`** in Java

- **Example**

```
AtomicInteger atomic = new AtomicInteger(5);
int nonAtomic = atomic.addAndGet(10);
// nonAtomic is now 15
```

Atomic data type – Live Demo

- **ThreadSynchronisation1 - atomicDatatypes**

Mutex Lock (1)

- A **mutex lock** (*abbr. for* mut*ual* ex*clusion*) takes care for only one thread entering a certain part of the code (***critical region***) at a time

- Example

```
ReentrantLock mutex = new ReentrantLock();
mutex.lock();
// do something useful }
mutex.unlock();
```

- The code between `lock()` and `unlock()` is executed by only one thread at a time

Mutex Lock – Live Demo

- **ThreadSynchronisation1 – mutexLock – reentrantLock**

Mutex Lock (2)

- **A mutex can also be used with a synchronized-block.**
  - A synchronized-Block needs an object as mutex
  - Also the this-object can server as mutex
  - All synchronized-Blocks, that share the same object, thus the object with the same memory address, belong together

- **Example**

```
SomeObject mutex = new SomeObject();
synchronized( mutex );
{
     // do something useful }
}
```

Mutex Lock (3)

```
public synchronized void func()
{
   // do something useful }
}
```

is the same as

```
public void func()
{
   synchronized(this)
   {
      // do something useful
   }
}
```

Mutex Lock – Live Demo

- **ThreadSynchronisation1 – mutexLock – synchronizedBlock**

Pipe

- A pipe (also called queue) is an uni- or bidirectional datastream, that works with the FIFO (*first in, first out*) principle

- Example

```
LinkedBlockingQueue < Integer > queue =
     new LinkedBlockingQueue < Integer >();

// Thread a
int t = queue.take (); // blocks if queue is empty

// Thread b
int p = 5;
queue.put(p)
```

Pipe – Live Demo

- **ThreadSynchronisation2 – pipe**

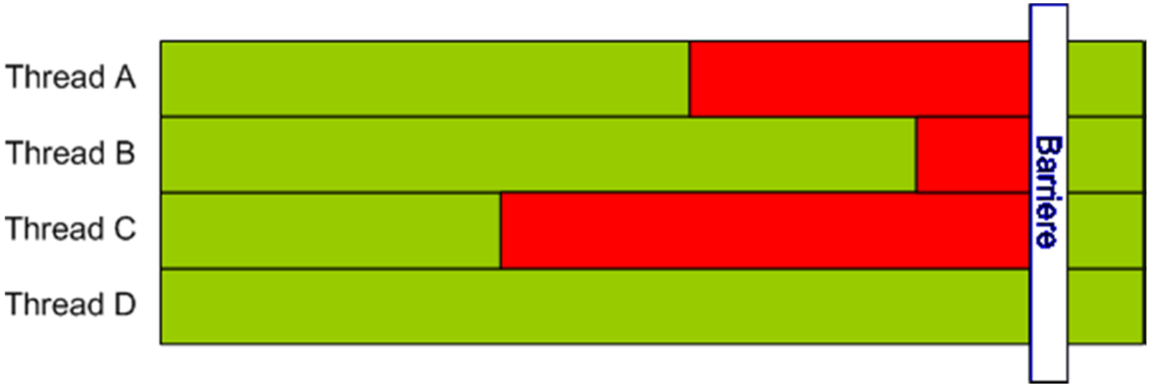
Barrier (1)

- **A barrier blocks all threads arriving at the barrier until a certain number of threads has reached the barrier**
  - The number of waiting threads is adjustable
  - When the last thread reaches the barrier, all threads are released
    - The barriers “breaks”.
- **Example**

```
int n = 4;
CyclicBarrier barrier = new CyclicBarrier(n);

try
{
    barrier.await();
}
catch( Exception e) { /* do something /* }
```

Barrier (2)



Barrier – Live Demo

- **ThreadSynchronisation2 – barrier**

Threadpool

- A future is an object which acts as placeholder for data, that will be available in the future

- Example

```
ExecutorService pool =
     Executors.newFixedThreadPool(5);

Callable <String > task = new TaskImplementation();
Future <String > f = pool.submit( task );
// Do something useful…
String result = f.get (); // blocks if necessary
```

Threadpool – Live Demo

- **ThreadSynchronisation2 – threadPool – runnables**

Future

- A threadpool is a group of threads
  - Each thread in the pool sleeps, until it gets a task
  - After finishing a task a thread returns to the pool
  - New tasks are queued if all threads are busy

- Example

```
ExecutorService pool =
     Executors.newFixedThreadPool (5);

Runnable task = new TaskImplementation();
pool.execute( task );
```

Future – Live Demo

- **ThreadSynchronisation2 – threadPool – futures**