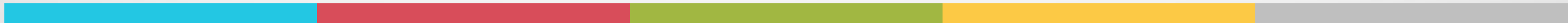
Threads

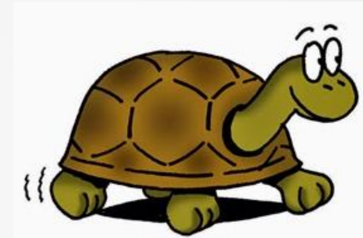


Geschichte der Systemarchitektur

- ✓ Anfänge der IT-Welt: DOS(Eine Ausführungseinheit z.B. nur ein Textprogramm welches die komplette Ein-Ausgabe übernahm)
- ✓ Heute: Mehrere Ausführungseinheiten, die sich die CPU-Leistung des Rechners nach Algorithmen verteilen, die im Betriebssystem implementiert sind
- ✓ Ein Prozess wird durch das Betriebssystem als eigenständige Instanz - in der Regel unabhängig und geschützt von anderen – ausgeführt (Multitaskingsysteme)
- ✓ Jeder Prozess hat seinen eigenen Stack und auch die Heapdaten sind im eigenen Adressraum vor dem Zugriff anderer Prozesse geschützt.
- ✓ Das Prozess-Scheduling des Betriebssystem sorgt dafür, dass die Rechnerleistung fair auf alle Prozesse verteilt wird
- ✓ Austausch von Daten zwischen den einzelnen Prozessen: IPC-Mechnismen (IPC = Inter Process Communication): Langsam und aufwändig zu kodieren

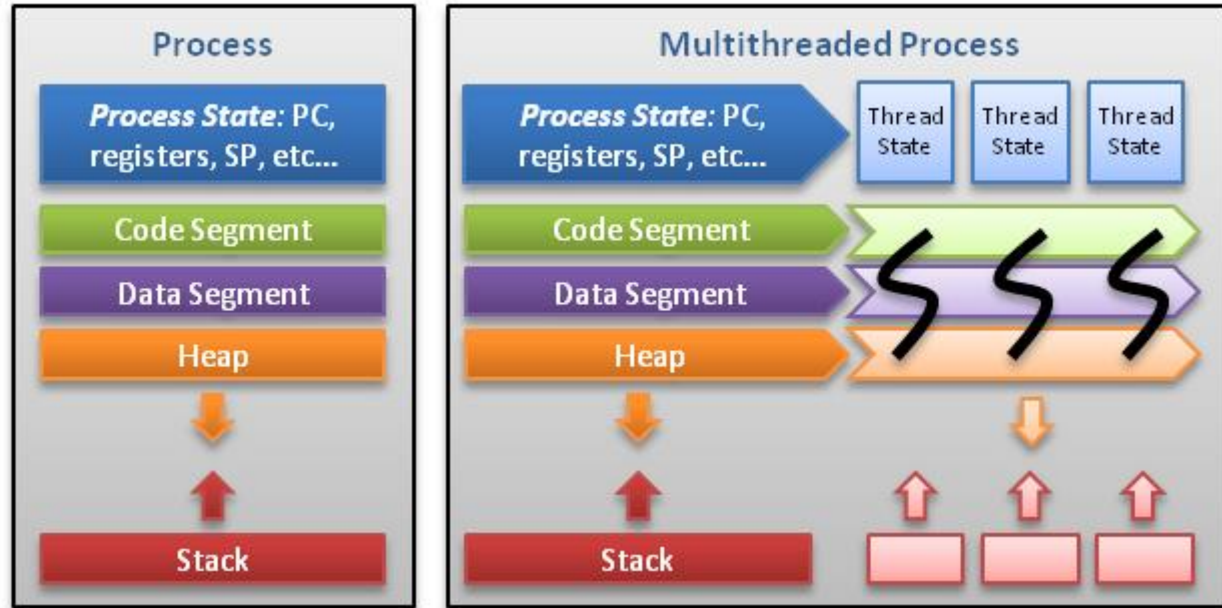
Threads - Motivation

Führt ein Programm eine lange Berechnung aus
z.B. eine Sortierung von 10000000 Objekten, dann
wird erst die nächste Anweisung ausgeführt, wenn diese
Zeilen ausgeführt sind



Besser: Methoden werden parallel abgearbeitet d.h.
diese laufen neben- und unabhängig voneinander!

Threads - Aufbau

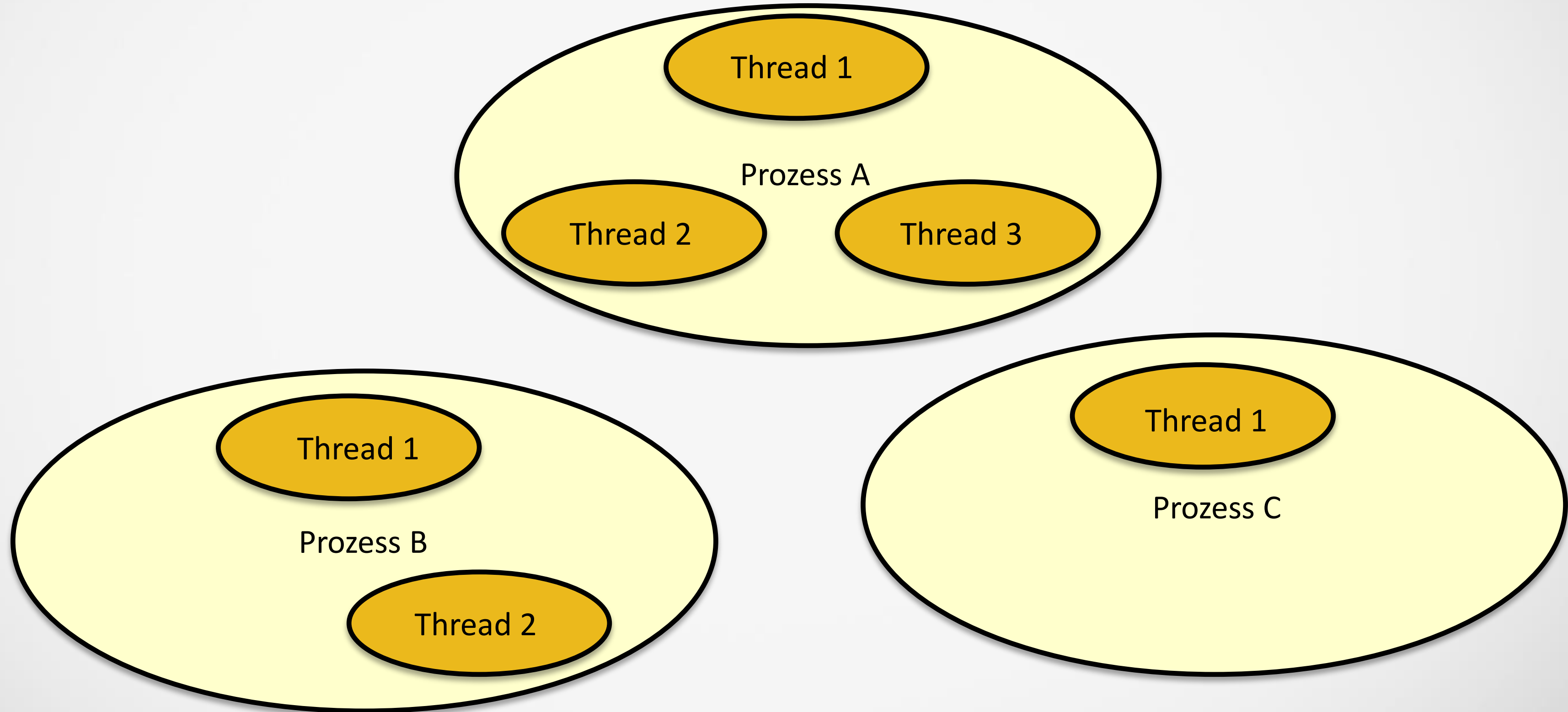


Ein Thread hat nur notwendige Informationen, wie Stack(Lokale Variablen, Funktionsargumente) und threadspezifische Daten. Alle anderen Daten werden gemeinsam genutzt

Prozesse vs Threads I

- ✓ Ein Prozess bezeichnet ein Programm in Ausführung mit seinen dafür notwendigen Datenstrukturen und zugeordneten Eigenschaften
- ✓ Ein Prozess wird durch das Betriebssystem als eigenständige Instanz - in der Regel unabhängig und geschützt von anderen – ausgeführt (Multitaskingsysteme)
- ✓ Ein Thread (engl. für Faden) stellt eine nebenläufige Ausführungseinheit innerhalb genau eines Prozesses dar, die parallel zu anderen Threads laufen kann.(-> feineren Ebene:“Leichtgewichtige Prozesse“)
- ✓ Nebenläufigkeit: Zwei Vorgänge gleichzeitig ausführen zu können

Prozesse vs Threads II



Erzeugung von Threads

- ✓ Thread stellt Basismethoden zur Verfügung und die Klasse muss davon abgeleitet werden
- ✓ Methode start: Starten des Threads
- ✓ Methode run: Bekommt von start die Methode run übertragen, so dass der Aufrufer direkt nach start fortfahren kann
- ✓ Die Methode run muss implementiert werden: Enthält die im Thread auszuführenden Anweisungen
- ✓ Die Methode run sollte vom Programm niemals direkt aufgerufen werden; statt dessen start

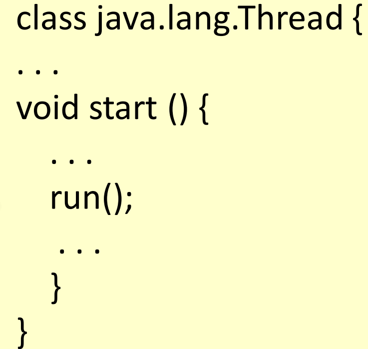
Erzeugung von Threads durch Ableiten von Thread

```
class MyThread extends Thread
{
  public void run()
  {
    // Aufgaben des neuen Threads
  }
}
```

```
MyThread neuerThread = new MyThread();
neuerThread.start();
//Alternativ:
//new MyThread().start();
```

Dateien->Ordner 01

Prinzip beim Ableiten der eigenen Klasse von Threads




```
class MyThread extends Thread {
void run() {
  // Programmcode
}
```

```
MyThread t = new myThread(..);
t.start();
```

Erzeugung von Threads durch Implementierung von Runnable

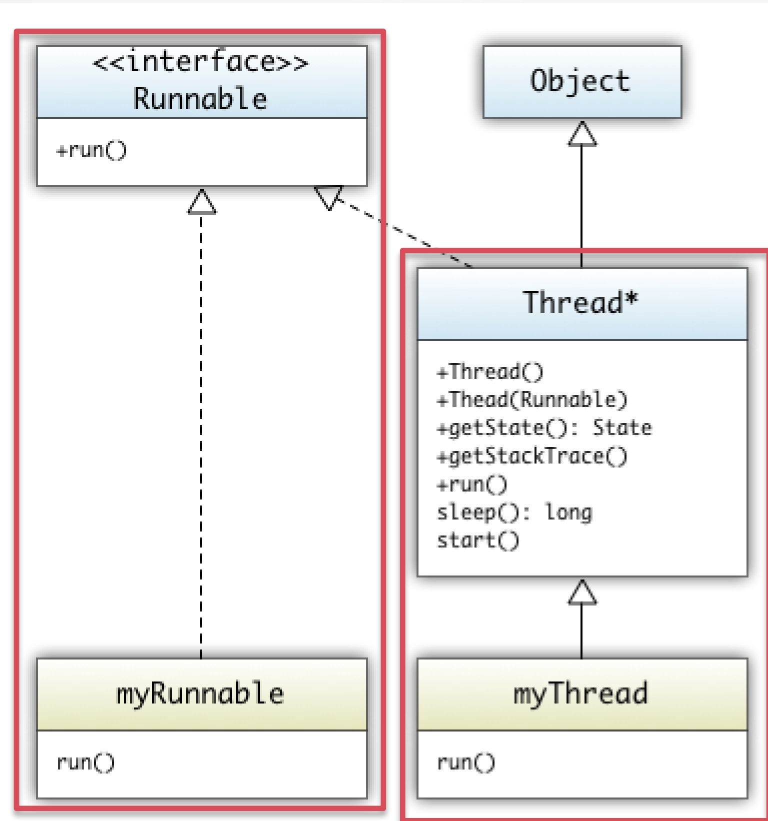
```
class MyThread implements Runnable
{
  public void run()
  {
    // Aufgaben des neuen Threads
  }
}
```

Klasse ist eine Implementierung von Runnable

Erzeugung eines Klassenobjektes und Starten von run

```
Thread neuerThread = new Thread(new MyThread());
neuerThread.start()
// Alternative:
// new Thread(new MyThread()).start();
```

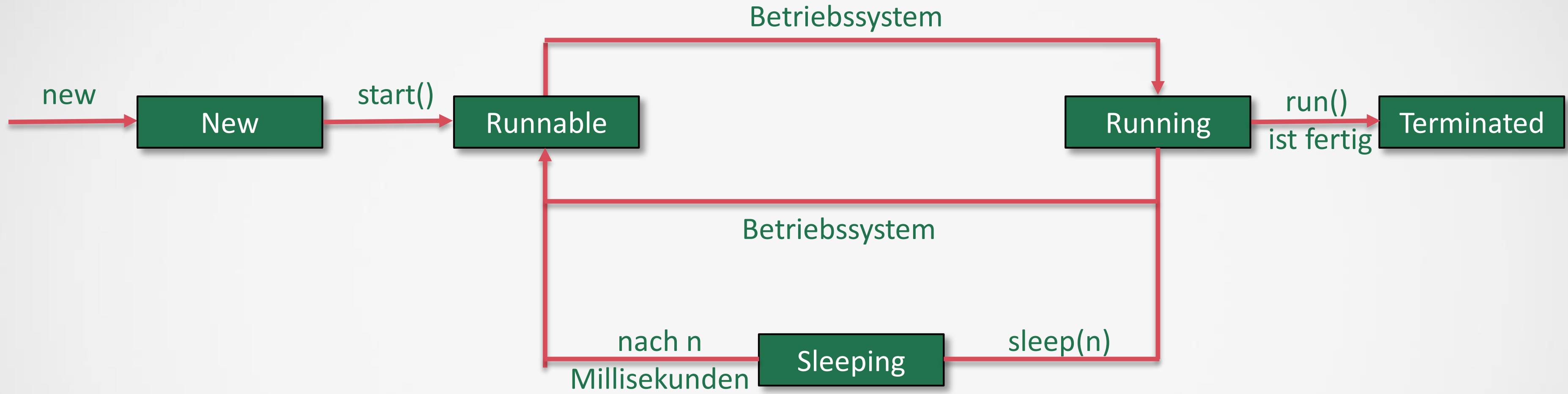
Übersicht über beide Varianten



Zustände eines Threads I

- **New:** MyThread neuerThread = new MyThread();
- **Runnable:** neuerThread.start();
- **Running:** Aktivierung des Thread durch den Scheduler, der dem Thread die CPU-Zeit zuteilt
- **Sleep:** Thread.sleep(1000);
oder Thread.yield();
oder Thread.join();
- **Terminated:** Wenn der Thread zu Ende ist oder durch interrupt()

Zustände eines Threads



Verwaltung von Threads

Threads können nicht nur ausgeführt und synchronisiert werden, sie besitzen auch noch zusätzliche administrative Eigenschaften:

- setName() (sonst Name = “Thread-"+n)
- getName()
- setPriority() (für Scheduler wichtig)
- getPriority()

“Schlafen” von Threads – Beispiel

MyThread wie im letzten Beispiel!

```
MyThread thread = new MyThread("My Thread");
  thread.start();
  System.out.println("Before Sleeping");
  try {
    thread.sleep(2000);
  }
  catch (InterruptedException e) {
    e.printStackTrace();
  }
  for (int i=0;i<200 ;i++ )
    System.out.println("no:"+i);
  System.out.println("After Sleeping");
```

Versetzt den Thread in 2000 Millisekunden in den Schlaf

Machen Sie eine Vorhersage über die Ausgabe!
Versuchen Sie, das abweichende Ergebniss zu verstehen (-> nächste Folie)!

Aussetzen von Threads – Beispiel

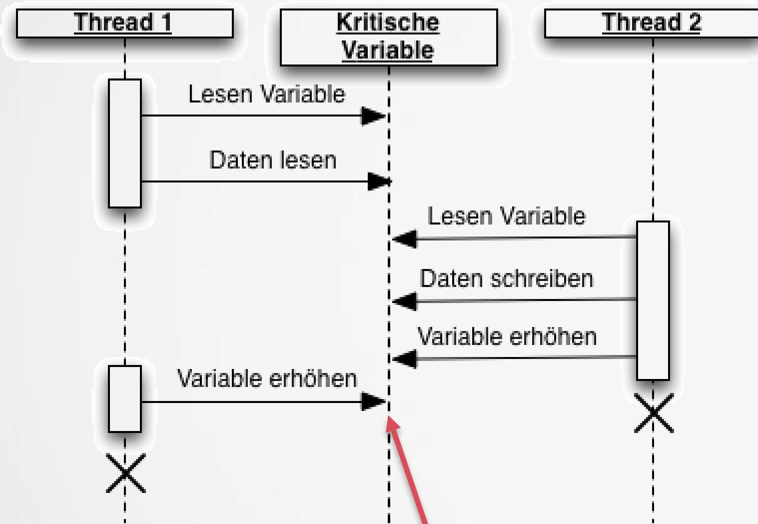
```
class MyThread extends Thread{
  @Override
  public void run(){
    for(int i = 0; i <= 10; i++){
         System.out.println(i);
         try {
              sleep(1000);
              }
         catch (InterruptedException e) {
              e.printStackTrace();
              }
         }
  }
}
```

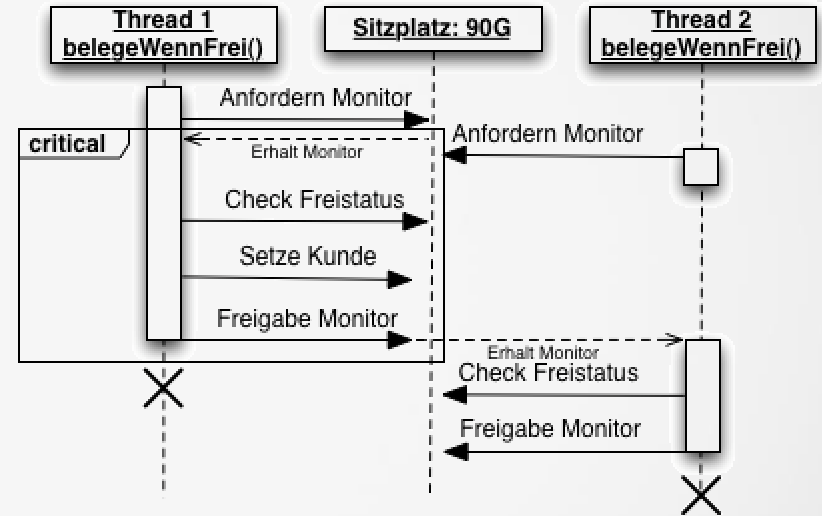
- sleep() versetzt immer den Thread, in dem er aufgerufen wurde, in den Schlaf! In diesem Fall also main()!
- Also muss sleep in die Klasse = Methode kodiert werden, wo der Aufruf des Threads erfolgt!
- Aufruf:

```
MyThread thread = new MyThread();
   thread.start();
   for (int i=1; i<=5; i++)
     System.out.println(“I print 1st!");
```

Probleme bei Threads:Buchungssystem

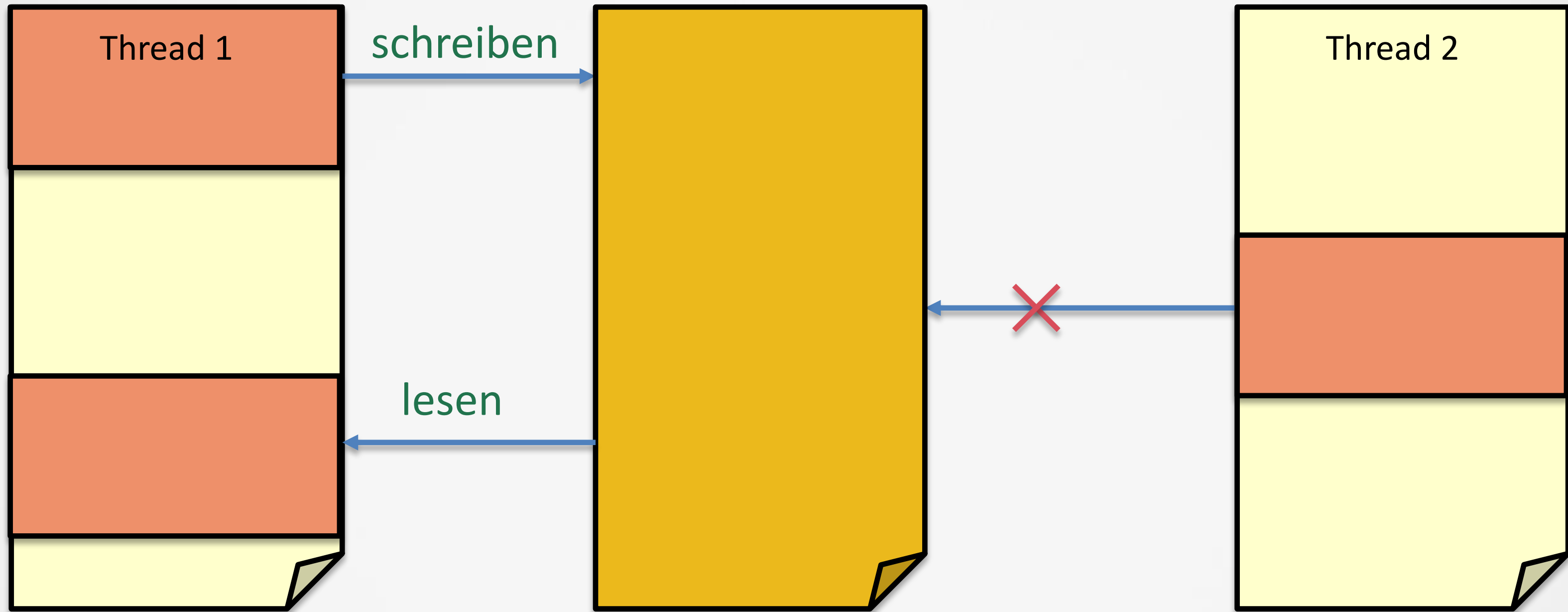
Probleme bei Threads: Buchungssystem - Lösung





Fehler

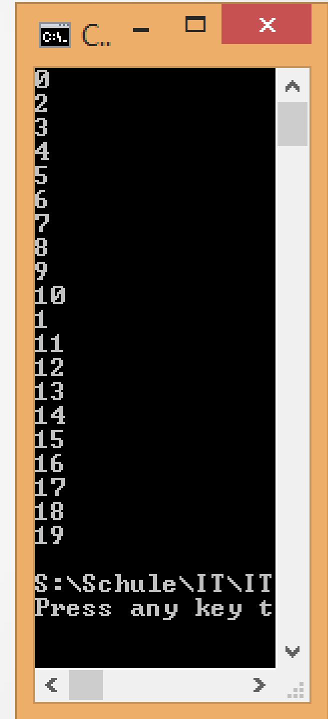
Prinzip der Synchronisation: Sperrung



Synchronisation von Threads – Motivation – Beispiel Ia

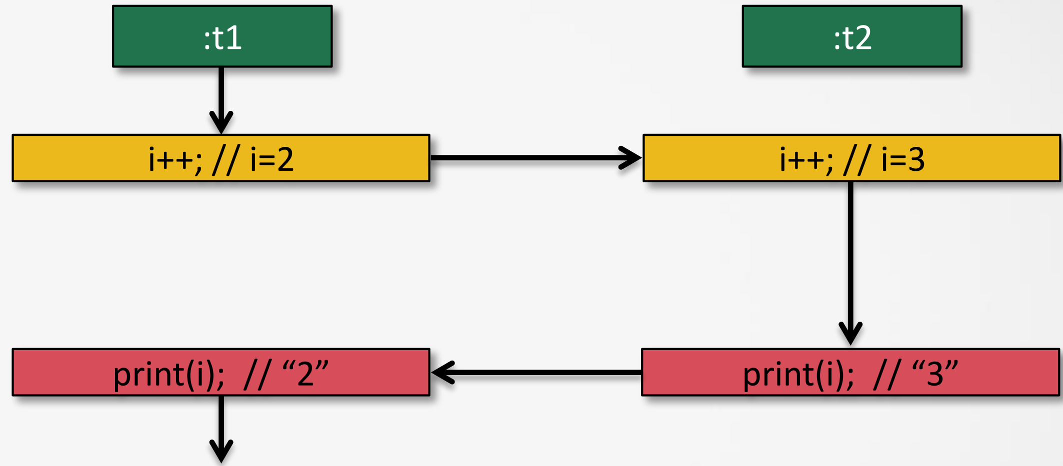
Die Kommunikation zweier Threads erfolgt in Java auf der Basis gemeinsamer Variablen

```
public class CSyncMotivation extends Thread {
  static int cnt = 0;
  public static void main(String[] args){
    Thread t1 = new CSyncMotivation();
    Thread t2 = new CSyncMotivation();
    t1.start();
    t2.start();
  }
  public void run(){
    int i=1;
    do {
      i++;
      System.out.println(cnt++);
    } while (i<10);
  }
}
```

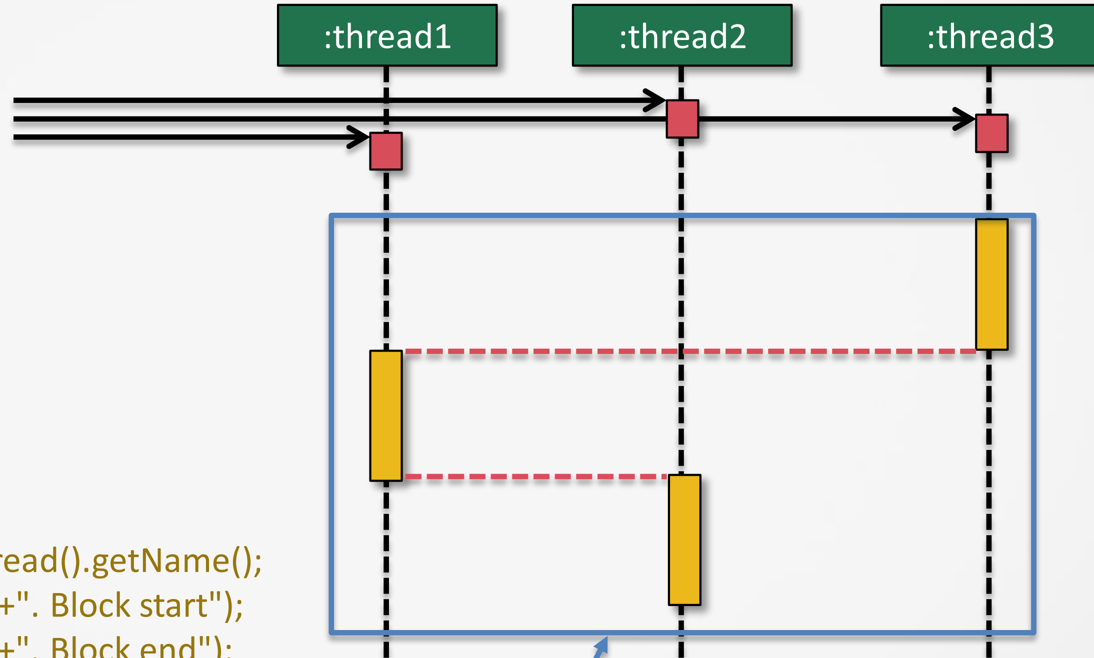


Synchronisation von Threads – Motivation – Beispiel Ib

```
do {
  i++;
  System.out.println(cnt++);
} while (i<10);
```



Pseudo-Sequenzdiagramm – Beispiel II



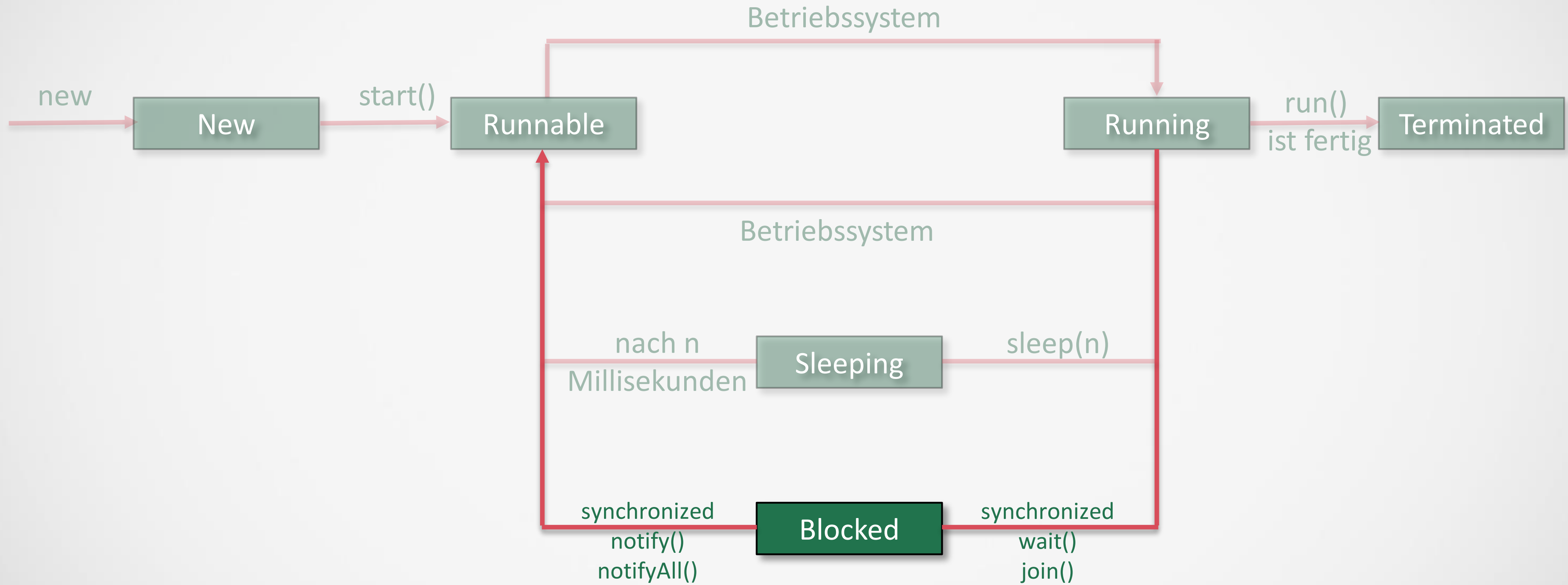

```
System.out.println(tname);
synchronized(getClass()){
    tname=Thread.currentThread().getName();
    System.out.println(tname+". Block start");
    System.out.println(tname+". Block end");
}
```

nicht überlappend

Synchronisation von Threads – Prinzip – Monitor

- Ein Monitor ist die Kapselung eines kritischen Bereichs (also eines Programmteils, der nur von jeweils einem Prozess zur Zeit durchlaufen werden darf) mit Hilfe einer automatisch verwalteten Sperre
- Jedes Objekt hat genau eine Sperre
- Diese Sperre wird beim Betreten des Monitors gesetzt und beim Verlassen wieder zurückgenommen
- Ist sie beim Eintritt in den Monitor bereits von einem anderen Prozess gesetzt, so muss der aktuelle Prozess warten, bis der Konkurrent die Sperre freigegeben und den Monitor verlassen hat.

Zustände eines Threads - Blockierung von Threads



Synchronisation von Threads – Erklärung I

```
class LockedOrNotLocked {
    synchronized void f() {
    // Dieses Objekt wird gesperrt
    …………
    // Dieses Objekt wird entsperrt
}
```

Synchronisation von Threads – Erklärung III

```
class LockedOrNotLocked {
    void f() {
        synchronized(this){
            // Dieses Objekt wird gesperrt
            ............
            // Dieses Objekt wird entsperrt
        }
}
```

Synchronized: Methodenblock

this: Dieses Objekt wird gesperrt

Diese Methode der Klasse wird synchronisiert für das aktuelle Objekt!

Synchronisation von Threads – Erklärung IV

```
class LockedOrNotLocked {
    void f() {
        synchronized(getClass()){
          // Diese Klasse wird gesperrt
          ............
         // Diese Klasse wird entsperrt
        }
}
```

Synchronized: Methodenblock
getClass(): Diese Klasse wird gesperrt

Diese Methode der Klasse wird synchronisiert für die aktuelle Klasse!

Synchronisation von Threads – Erklärung II

```
class LockedOrNotLocked {
    synchronized static void f() {
    // Diese Klasse wird gesperrt
    …………
    // Diese Klasse wird entsperrt
}
```

Synchronized: Klasse

Diese Methode der Klasse wird synchronisiert für die aktuelle Klasse!

Synchronisation von Threads – Gleichwertige Varianten von Object Lock

```
class LockedOrNotLocked {
    synchronized void f() {
    // Dieses Objekt wird gesperrt
    …………
    // Dieses Objekt wird entsperrt
}
```

```
class LockedOrNotLocked {
    void f() {
        synchronized(this){
            // Dieses Objekt wird gesperrt
            …………
            // Dieses Objekt wird entsperrt
        }
}
```

Sperrt das jeweilige Objekt

Synchronisation von Threads – Gleichwertige Varianten von Class Lock

```
class LockedOrNotLocked {
    void f() {
      synchronized(getClass()){
        // Diese Klasse wird gesperrt
        …………
       // Diese Klasse wird entsperrt
      }
}
```

```
class LockedOrNotLocked {
    synchronized static void f() {
    // Diese Klasse wird gesperrt
    …………
    // Diese Klasse wird entsperrt
}
```

Sperrt die jeweilige Klasse

Synchronisation von Threads – Regeln II

Konstruktoren oder dynamischer oder statischer initialisierter Quelltext dürfen nicht synchronisiert werden, aber synchronisierten Quelltext enthalten:

```
public class Counter{
  synchronized int i;
  synchronized public Counter(){……}
  synchronized static  {….}
  synchronized {….}
}
```

Jede Zeile ergibt einen Fehler!

Korrekte Version

```
public class Counter{
  public Counter(){
    synchronized (this){…..}
  }
  static {
    synchronized (Counter.class) {…..}
  }
  {
    synchronized (this) {…..}
  }
}
```

Synchronisation von Threads – Beispiel I

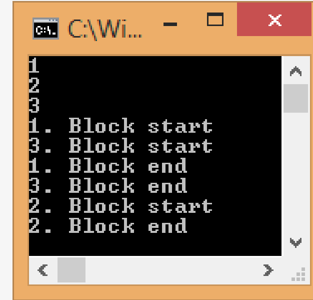
```
class Synchronize_Objects implements Runnable {
   public void run(){
      Lock();
   }
   public void Lock(){
      String tname=Thread.currentThread().getName();
      System.out.println(tname);
      synchronized(this){
         tname=Thread.currentThread().getName();
         System.out.println(tname+". Block start");
         System.out.println(tname+". Block end");
      }
   }
```

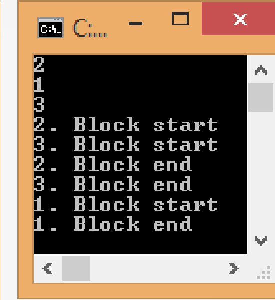
```
public static void main(String[] args)
   {
      Synchronize_Objects object1 = new Synchronize_Objects();
      Synchronize_Objects object2 = new Synchronize_Objects();
      Thread myThread1 = new Thread(object1);
      Thread myThread2 = new Thread(object1);
      Thread myThread3 = new Thread(object2);
      myThread1.setName("1");
      myThread2.setName("2");
      myThread3.setName("3");
      myThread1.start();
      myThread2.start();
      myThread3.start();
   }
}
```

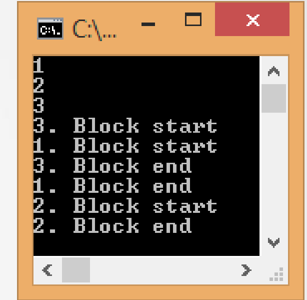
Synchronisation von Threads – Beispiel Ib

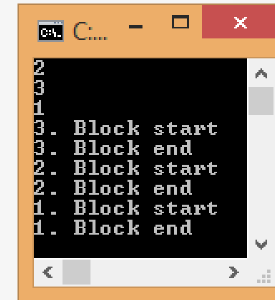
```
Thread myThread1 = new Thread(object1);
Thread myThread2 = new Thread(object1);
Thread myThread3 = new Thread(object2);
```

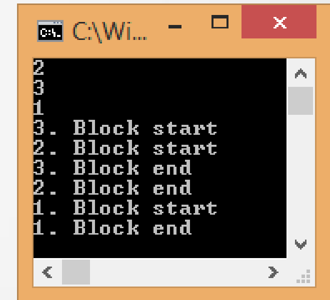
```
System.out.println(tname);
synchronized(this){
   tname=Thread.currentThread().getName();
   System.out.println(tname+". Block start");
   System.out.println(tname+". Block end");
}
```







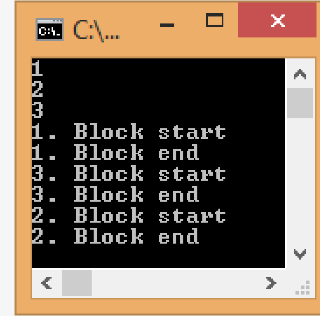


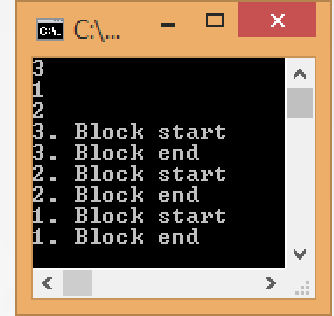


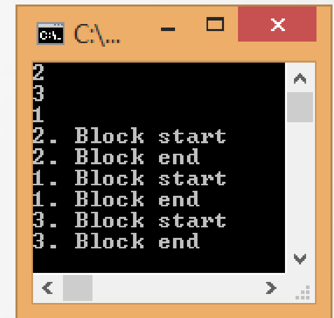
Synchronisation von Threads – Beispiel II

```
Thread myThread1 = new Thread(object1);
Thread myThread2 = new Thread(object1);
Thread myThread3 = new Thread(object2);
```

```
System.out.println(tname);
synchronized(getClass()){
   tname=Thread.currentThread().getName();
   System.out.println(tname+". Block start");
   System.out.println(tname+". Block end");
}
```

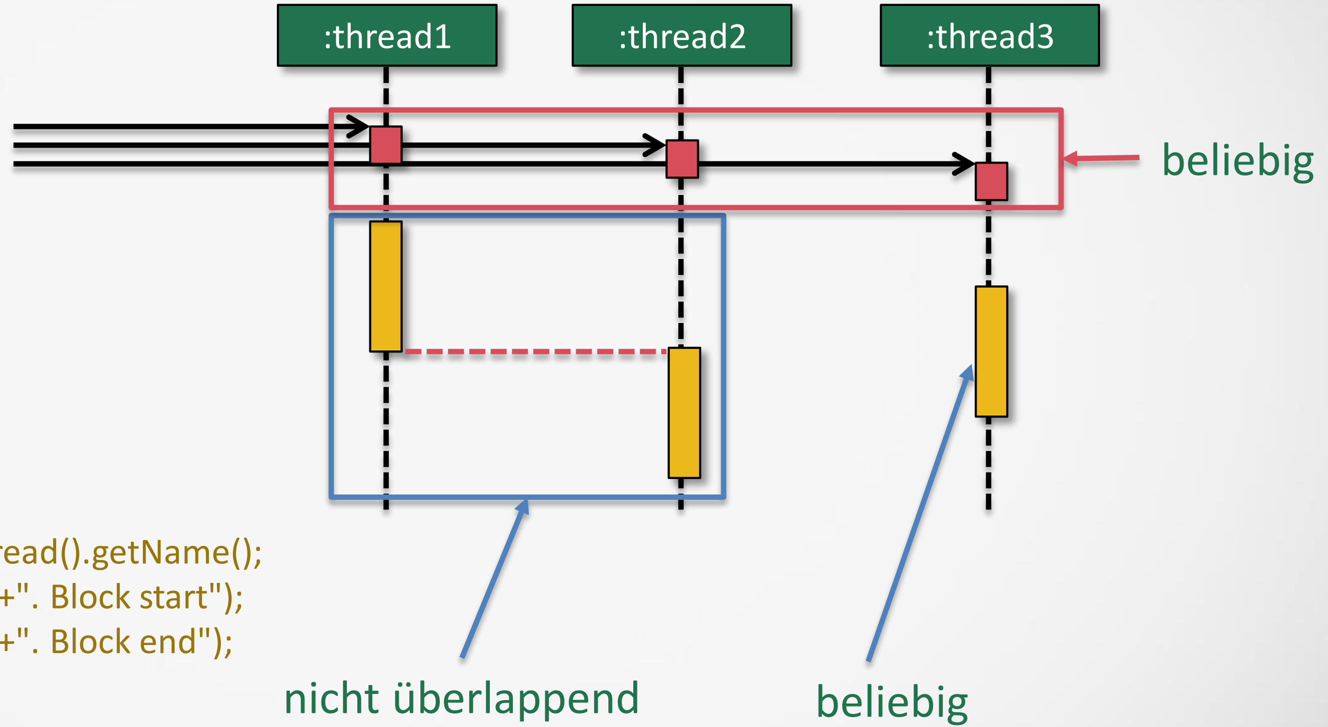






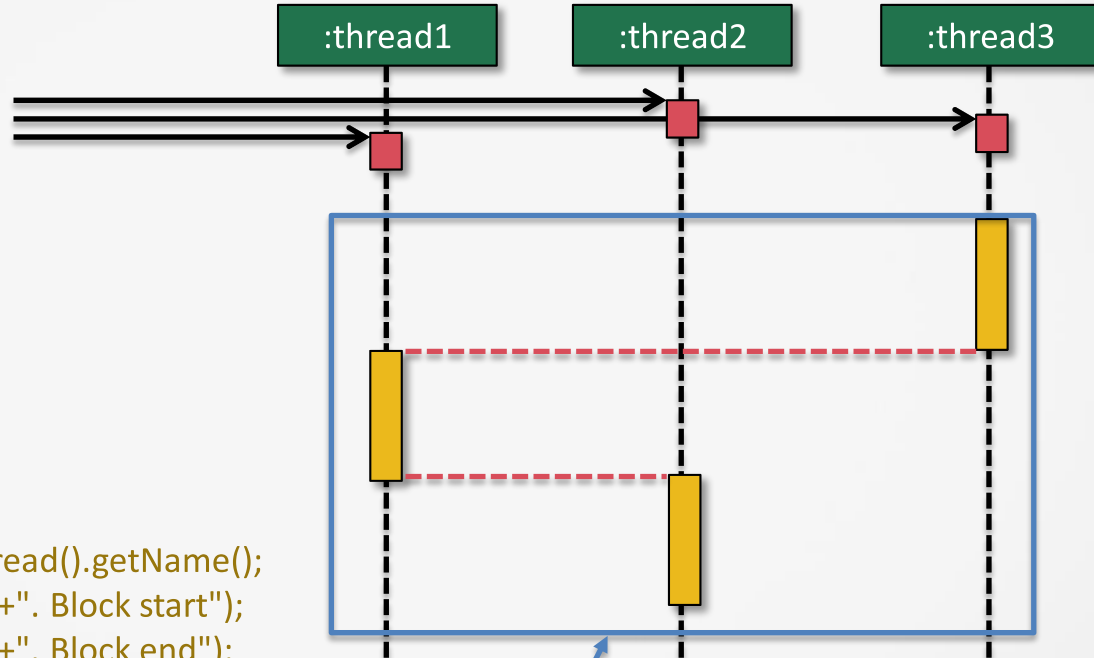
getClass() ist mit Synchronize_Objects.class gleichwertig!

Pseudo-Sequenzdiagramm – Beispiel I




```
System.out.println(tname);
synchronized(this){
   tname=Thread.currentThread().getName();
   System.out.println(tname+". Block start");
   System.out.println(tname+". Block end");
}
```

Pseudo-Sequenzdiagramm – Beispiel II




```
System.out.println(tname);
synchronized(getClass()){
   tname=Thread.currentThread().getName();
   System.out.println(tname+". Block start");
   System.out.println(tname+". Block end");
}
```

nicht überlappend

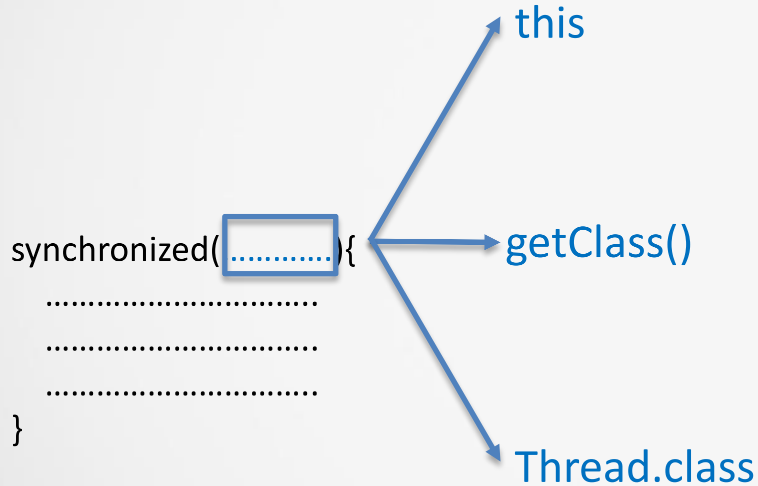
Synchronisation von Threads – Beispiel III

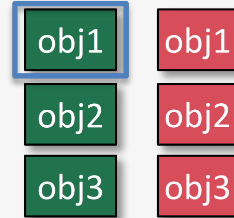
```
TestLock object1 = new TestLock();
TestLock2 object2 = new TestLock2();
Thread myThread1 = new Thread(object1);
Thread myThread2 = new Thread(object1);
Thread myThread3 = new Thread(object2);
```

```
System.out.println(tname);
synchronized(getClass()){
   tname=Thread.currentThread().getName();
   System.out.println(tname+". Block start");
   System.out.println(tname+". Block end");
}
```

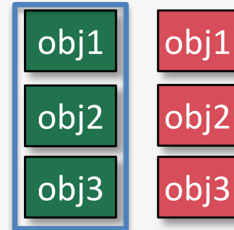
Mit getClass() werden alle laufenden Threads klassenweise synchronisiert!
Mit Thread.class() werden alle laufenden Threads global(für das gesamte Programm) synchronisiert!

Synchronisation von Threads – Übersicht





Nur ein Objekt der drei Objekte der Klasse

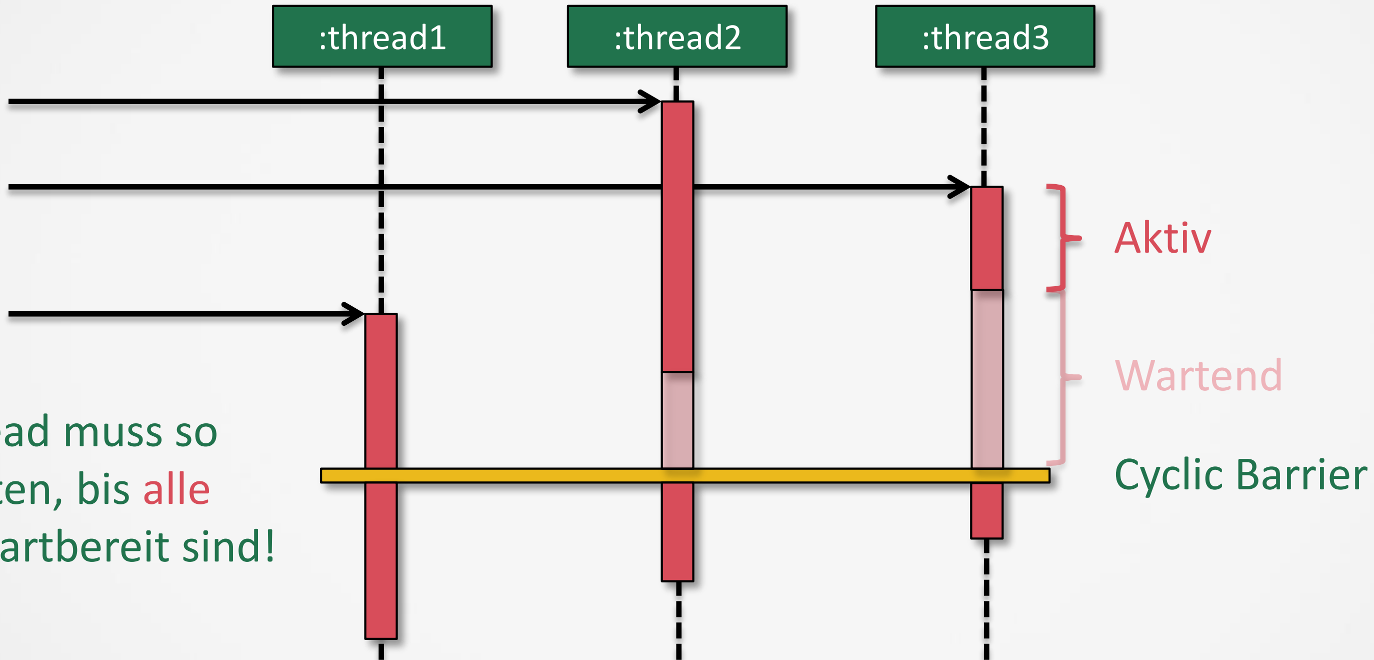


Alle Objekte der drei Objekte der Klasse

Alle Objekte aller Klassen(damit aller Threads)

Anwendungen von Threads: Cyclic Barrier

Jeder Thread muss so lange warten, bis alle Threads startbereit sind!



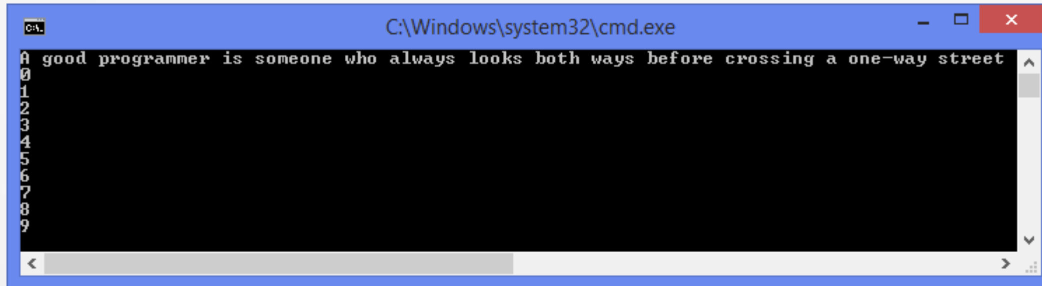
Beispiel: Multiplayer Spiel, wo alle X Spieler eingeloggt sein müssen!

Konstruktoren von der Thread – Klasse

Thread(): It creates a Thread object with a default name.
Thread(String name): It creates a Thread object with a name that the name argument specifies.
Thread (Runnable target): This method constructs Thread with a parameter of the Runnable object that defines the run() method.
Thread (Runnable target, String name): This method creates Thread with a name and a Runnable object parameter to set the run() method

Reihenfolge des Starten von Threads – Beispiel a

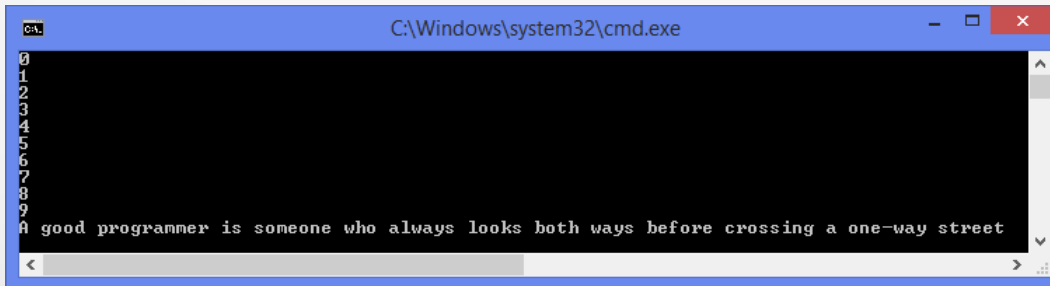
```
public class ThreadsInJava
{
  public static void main(String[] args)   {
    MyThread myThread = new MyThread();
    myThread.start();
    System.out.println("A good programmer…");
  }
}
```



Reihenfolge des Starten von Threads – Beispiel b

```
public class ThreadsInJava
{
  public static void main(String[] args)   {
    MyThread myThread = new MyThread();
    myThread.start();
    try{
      myThread.join();
    }
    catch(InterruptedException ex)
    {}
    System.out.println("A good programmer…");
  }
}
```

Join zwingt den aktuellen Thread(=main) mit der Ausführung zu warten bis, dieser fertig ist!



Angebot des Abgebens der Zeitscheibe – Beispiel

```
class MyThread extends Thread{
  @Override
  public void run(){
    for(int i = 0; i < 100; i++){
      System.out.println(i);
      this.yield();
    }
  }
}
```

Yield bietet dem Scheduler das Abgeben der Zeitscheibe an – der Scheduler kann dies annehmen oder ignorieren!

yield wird intern mit sleep(0) ausgeführt, d.h. der Thread stellt sich hinten in die Warteschlange aller warteten Threads mit gleicher Priorität an!