Programmierung 2

DHBW Mannheim - WWI17SCB - May 2018

Slides

https://goo.gl/U45wY3

6. Threads

Threads - Lernziele

* Unterschied zwischen Multi-Tasking und Multi-Threading erklären
* Wesentliche Anwendungsgebiete von Threads kennen
* Threads in Java erzeugen, starten und beenden
* Unterschiedliche Zustände eines Threads kennen und erläutern
* Threads beim Zugriff auf gemeinsame Daten synchronisieren
* Das Consumer-Producer-Problem kennen und implementieren können

Was sind Threads?

Was sind Threads?

* Moderne Betriebssysteme scheinen Programme parallel auszuführen --> Multi-Tasking
* Multi-Tasking: bei Rechnern mit einem Prozessor wird zwischen **Prozessen** (Programme, Daten in festem Speicheradressraum, Ressourcen) in kurzen Zeitabständen umgeschaltet
* Umschaltung wird vom Scheduler des OS übernommen
* Anwender nimmt eine Quasi-Parallelität der Abarbeitung wahr

--> Parallelität von Programmen auf Betriebssystemebene

Was sind Threads?

* Auf Mehrkern-CPUs können durch paralleles Arbeiten Performancevorteile erreicht werden
* Ziel: Quasi-Parallelität nicht nur auf Betriebssystemebene, sondern innerhalb eines Programms zur Verfügung zu haben
* Jeder CPU-Kern führt einen separaten Teil eines Prozesses aus
* Diese Teile heißen **Threads**

Was sind Threads?

* Threads laufen innerhalb eines Prozesses und somit innerhalb eines festen Speicherbereichs
* Zugriff auf öffentliche Attribute (alle Ressourcen des Prozesses) möglich
* Falls das Betriebssystem kein Multi-Threading unterstützt, simuliert die JVM die Parallelität (In Desktop-Umgebungen ist Multi-Threading Standatd embedded nicht unbedingt!)

Activity Monitor (My Processes)

Prozesse

CPU | Memory | Energy | Disk | Network | Search

Threads pro Prozess

Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
Google Chrome Helper	9,6	1:43:15,29	26	[illegible]	[illegible]	bastian
Activity Monitor	7,8	2,02	11	2	4408	bastian
Google Chrome	7,7	42:35,31	46	82	1139	bastian
Slack	1,6	34,03	32	2	2908	bastian
Google Chrome Helper	1,3	13:25,50	29	8	1582	bastian
Eclipse	0,9	5:42,95	58	52	3862	bastian
Google Chrome Helper	0,8	14:48,36	9	4	1174	bastian
activeCollabTimer3	0,4	1:30,79	8	14	1141	bastian
IntelliJ IDEA	0,3	22:04,70	52	33	1422	bastian
Dock	0,2	8,13	4	7	1143	bastian
Dropbox	0,1	1:10,86	121	1	1586	bastian
Google Chrome Helper	0,1	1:01,01	17	1	1570	bastian
java	0,1	5,93	24	19	4116	bastian
iTerm2	0,1	42,47	9	2	1138	bastian
Slack Helper	0,1	53,24	19	1	2914	bastian
fontd	0,0	6,13	3	0	1062	bastian
Slack Helper	0,0	11,36	19	1	2919	bastian
Slack Helper	0,0	16,90	5	0	2909	bastian
ViewBridgeAuxiliary	0,0	0,49	2	1	4335	bastian
cfprefsd	0,0	3,43	9	0	497	bastian
SystemUIServer	0,0	4,22	4	0	1144	bastian
SafariBookmarksSyncAgent	0,0	4,51	4	1	1118	bastian
mysqld	0,0	5,91	30	4	1314	bastian

		CPU LOAD		
System:	2,67%		Threads:	1689
User:	4,89%		Processes:	359
Idle:	92,44%			

Hide slides

Prozesse vs Threads

Prozesse

* Abgeschlossene Ausfühungs-umgebung (Speicher)
* Kein direkter Zugriff auf Ressourcen eines anderen P.
* Kommunikation mit anderen P. über Pipes/Sockets
* Betriebssystemebene

Threads

* Geteilte Ausführungs-umgebung mit anderen Threads
* Direkte Zugriffsmöglichkeit auf gemeinsame Ressourcen
* Kommunikation mit anderen T. via Interrupt / **Thread**-Methoden
* Programmebene

Anwendungsgebiete von Threads

* Gleichzeitige Nutzung unterschiedlicher Hardware-Ressourcen
  * Hauptspeicherzugriffe
  * Prozessor
  * Dateioperationen
  * Festplatte
  * Datenbankzugriff
  * Server, Netzwerkverbindung
* Steigerung der Verarbeitungsgeschwindigkeit

Anwendungsgebiete von Threads

Beispiele: Gleichzeitiges...

* Öffnen eines Fensters und Öffnen bzw. Lesen einer Datei
* Reagieren auf Benutzereingaben und Laden einer Internet-URL
* Lesen von Daten aus Datei und Analyse bereits geladener Daten
* Analyse neu gelesener Daten und Speichern von Daten in DB
* Typisches Beispiel: Consumer-Producer-Problematik

Ihr erinnert euch vielleicht an den LaggyActionListener!

Threads erzeugen

Das Interface Runnable

* **Runnable** kapselt einen so genannten Task - eine ausführbare Einheit, die einem Thread zur Abarbeitung übergeben werden kann
* Beispiel:

```
1 Runnable task1 = new Runnable() {
2   public void run() {
3     for (int i = 0; i <= 10; i++) {
4       System.out.println(i);
5     }
6   }
7 };
```

Das Interface Runnable

* Ein Runnable kann direkt aufgerufen werden - damit haben wir aber noch keinen Thread erzeugt!

```
1 Runnable task1 = new Runnable() {
2   public void run() {
3     for (int i = 0; i <= 10; i++) {
4       System.out.println(i);
5     }
6   }
7 };
8 task1.run();
```

```
1 0
2 1
3 2
4 ...
5 9
6 10
7
8 Process finished with exit code 0
```

Erzeugen von Threads

* Zur Erzeugung eines Threads übergeben wir das Runnable und rufen start() auf.

```
1 Runnable task1 = new Runnable() {
2     public void run() {
3     for (int i = 0; i <= 10; i++) {
4       System.out.println(i);
5     }
6   }
7 };
8 Thread t1 = new Thread(task1);
9 t1.start();
```

```
1 0
2 1
3 2
4 ...
5 9
6 10
7
8 Process finished with exit code 0
```

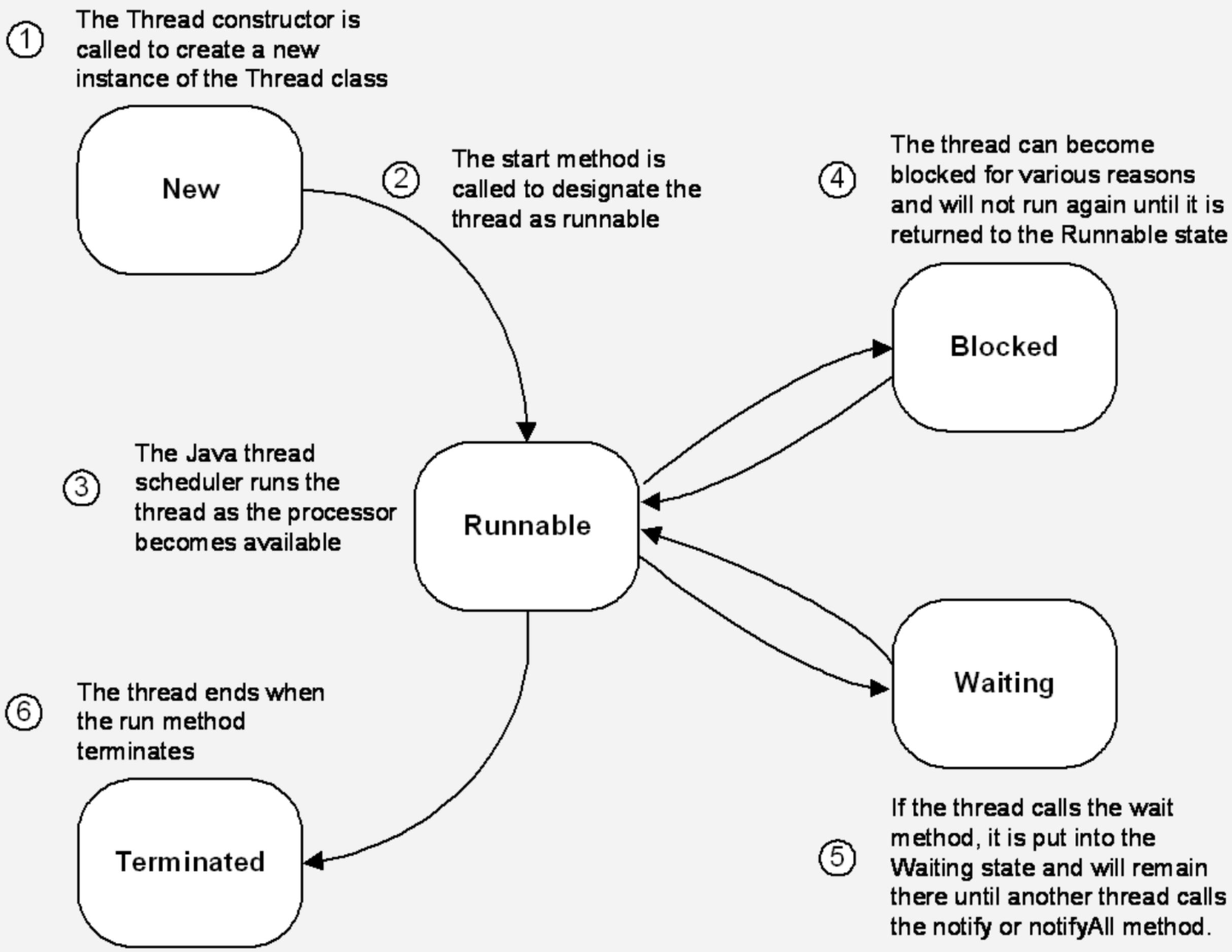
Erzeugen von Threads

* Nun können wir zusätzlich einen weiteren Thread starten.
* Beide Threads laufen parallel!
* Quizfrage: Wie viele Threads laufen nun insgesamt?

```
1 Runnable task2 = new Runnable() {
2     public void run() {
3     for (int i = 10; i >= 0; i++) {
4       System.out.println(i);
5     }
6   }
7 };
8 Thread t2 = new Thread(task2);
9 t2.start();
```

```
1 0
2 1
3 10
4 9
5 8
6 7
7 2
8 3
9 ...
```

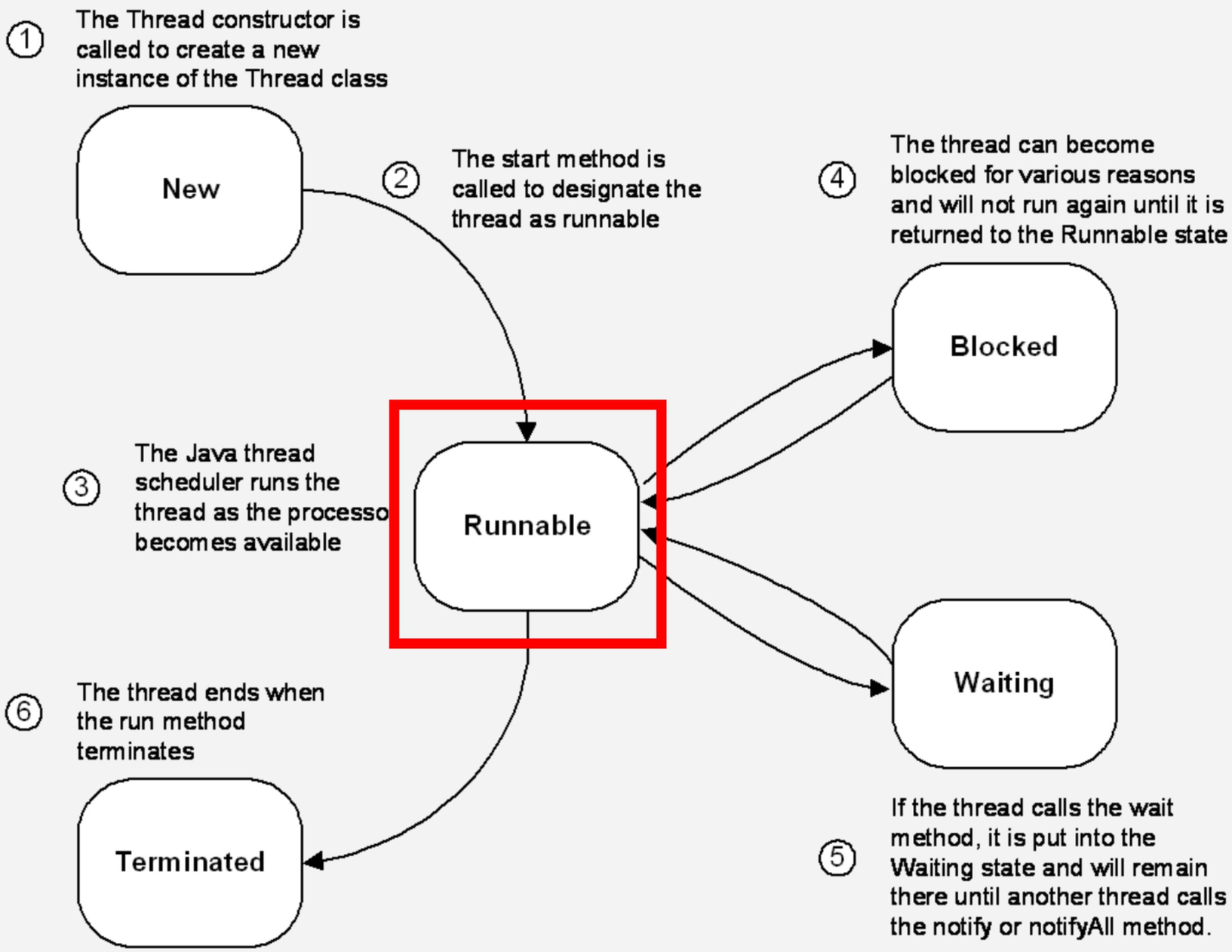
Zustände von Threads



http://voyager.deanza.edu/~hso/cis35a/lecture/java14/intro/lifecycle.html

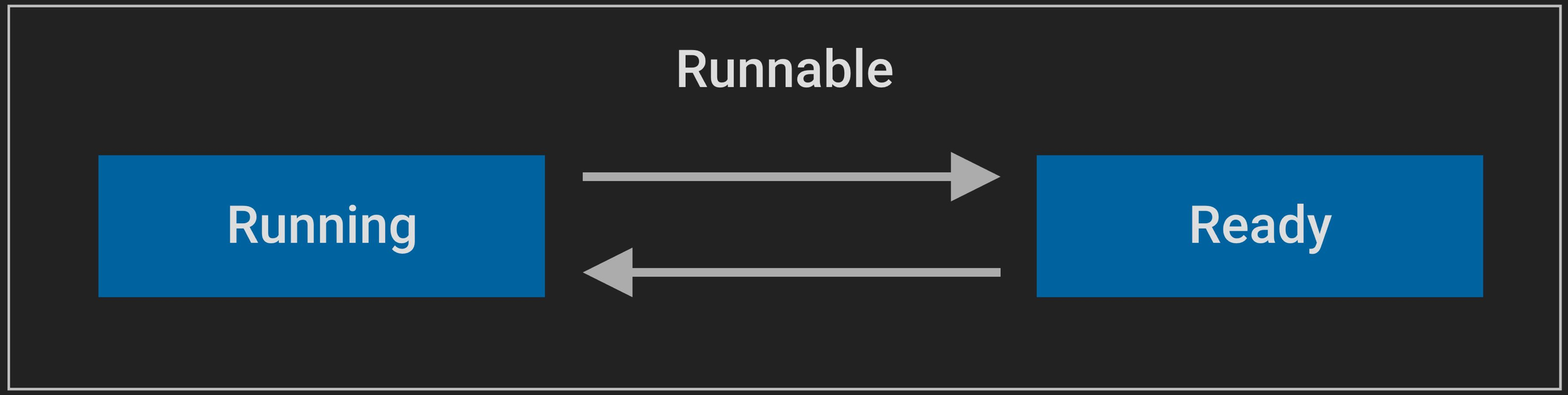
Hinweise zum Zustand

* Nach dem Aufruf von start() ist es dem Scheduler des OS überlassen, ob er einen Thread tatsächlich laufen lässt
* Threads im Zustand "Runnable" können beliebig oft zwischen den inneren Zuständen "Running" und "Ready" hin- und herschalten
* Durch Warten auf blockierende Systemressourcen oder Aufruf von sleep() wird ein Thread geblockt ("Blocked")
* Threads können auch von außen durch Interrupts geblockt werden
* Durch Aufruf von wait() wird ein Thread "Waiting"



http://voyager.deanza.edu/~hso/cis35a/lecture/java14/intro/lifecycle.html

Runnable heißt nicht Running!

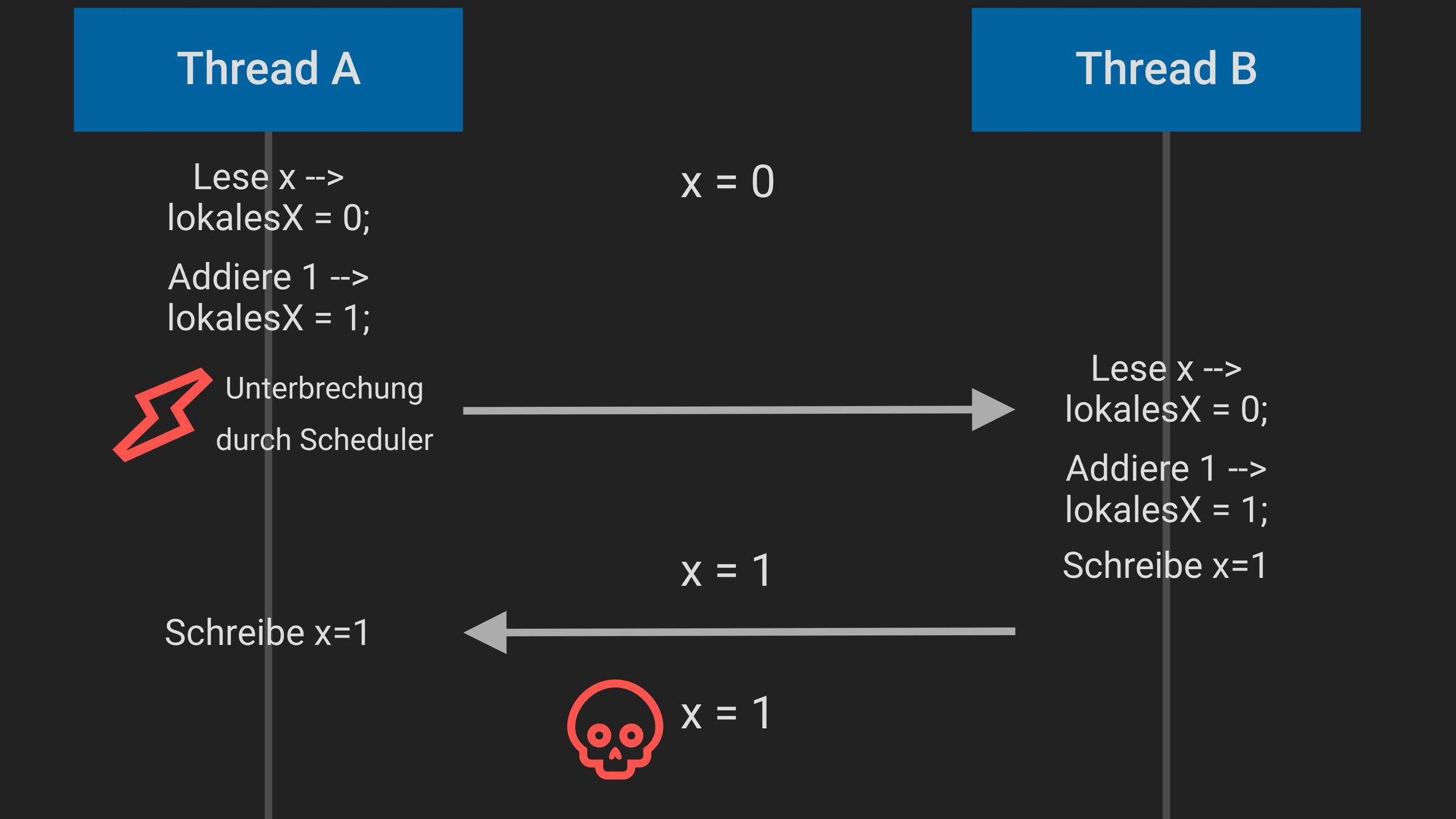


* Im Zustand Runnable kann ein Thread zu jedem beliebigen Zeitpunkt entweder laufen ("Running") oder bereit sein ("Ready")
* Die Entscheidung darüber liegt **ausschließlich beim Betriebssystem!**

Gemeinsame Datenzugriffe

Zugriffe auf gemeinsame Daten

* Wenn Threads auf gemeinsame Daten zugreifen, können Probleme entstehen. Beispiel:
* Thread A liest Wert x=0, addiert 1 und will x=1 schreiben
* Zwischen Lesen und Schreiben wird Thread A von B unterbrochen.
* Thread B liest Wert x=0, addiert 1 und schreibt x=1 zurück
* Thread A ist wieder dran und schreibt x=1 zurück
* x wurde nun zweimal inkrementiert, aber hat am Ende den Wert 1.
* Es ist ein "**Memory Consistency Error**" entstanden.


Thread A
Thread B
Lese x --> lokalesX = 0;
x = 0
Addiere 1 --> lokalesX = 1;
Unterbrechung durch Scheduler
Lese x --> lokalesX = 0;
Addiere 1 --> lokalesX = 1;
Schreibe x=1
x = 1
Schreibe x=1
x = 1

Speicherkonsistenzfehler - Codebeispiel

```
1 public class MemoryConsistency1 {
2     static int x = 0;
3
4     static int getX() {
5         return x;
6     }
7
8     static void setX(int newX) {
9         x = newX;
10    }
11
12    ... more code here, see next slide
13 }
```

Speicherkonsistenzfehler - Codebeispiel

```
1 private static class Counter extends Thread {
2   public void run() {
3     while(true) {
4       int localX = getX();
5       System.out.println(getName() + " read: " + localX);
6       localX++;
7       setX(localX);
8       System.out.println(getName() + " wrote: " + localX);
9       try {
10        sleep(10);
11      } catch (InterruptedException e) {
12        //
13      }
14    }
15  }
16 }
```

Speicherkonsistenzfehler - Codebeispiel

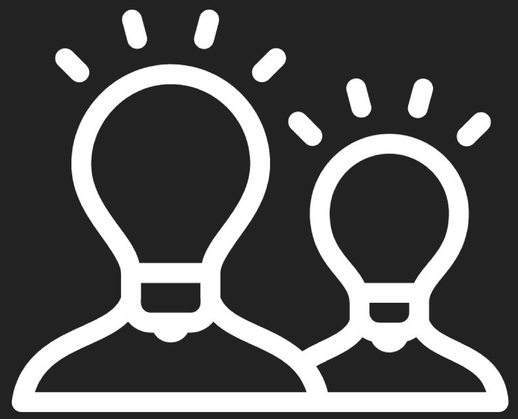
```
1 public static void main(String[] args) {
2   new Counter().start();
3   new Counter().start();
4   new Counter().start();
5   new Counter().start();
6 }
```

```
1 Thread-0 read: 0
2 Thread-0 wrote: 1
3 Thread-2 read: 0
4 Thread-2 wrote: 1
5 Thread-1 read: 0
6 Thread-1 wrote: 1
7 Thread-3 read: 1
8 Thread-3 wrote: 2
9 Thread-0 read: 2
10 Thread-0 wrote: 3
```

Speicherkonsistenzfehler lösen

* Lösungsversuch 1: Atomare Funktionen verwenden!

--> "Wenn ich keine lokale Variable im Thread habe, dürfte ja nix schief gehen!"



Speicherkonsistenzfehler - Codebeispiel

```
1 public class MemoryConsistency1 {
2     static int x = 0;
3
4     static int incX(){
5         return x++;
6     }
7
8     ... more code here, see next slide
9 }
```

Speicherkonsistenzfehler - Codebeispiel

```
1 private static class Counter extends Thread {
2   public void run() {
3     while(true) {
4       int localX = incX();
5       System.out.println(getName() + " wrote: " + localX);
6       try {
7         sleep(10);
8       } catch (InterruptedException e) {
9         //
10      }
11    }
12  }
13 }
```

Speicherkonsistenzfehler - Codebeispiel

```
1 public static void main(String[] args) {
2   new Counter().start();
3   new Counter().start();
4   new Counter().start();
5   new Counter().start();
6 }
```

Sieht gut aus...

```
1 Thread-0 wrote: 0
2 Thread-2 wrote: 2
3 Thread-3 wrote: 3
4 Thread-1 wrote: 1
5 Thread-2 wrote: 4
6 Thread-3 wrote: 5
7 Thread-0 wrote: 6
8 Thread-1 wrote: 7
9 ...
```

Speicherkonsistenzfehler - Codebeispiel

```
1 public static void main(String[] args) {
2   new Counter().start();
3   new Counter().start();
4   new Counter().start();
5   new Counter().start();
6 }
```

...oder nicht.

```
1 Thread-2 wrote: 8
2 Thread-3 wrote: 8
3 Thread-0 wrote: 8
4 Thread-1 wrote: 8
5 Thread-0 wrote: 9
6 Thread-3 wrote: 9
7 Thread-2 wrote: 9
8 Thread-1 wrote: 9
9 Thread-2 wrote: 10
```

Speicherkonsistenzfehler lösen

* Warum geht das nicht? "x++" ist doch eine einzelne Operation - wie kann ein Thread darin unterbrochen werden?
* Antwort: Für die JVM sind das mehrere Befehle!

1. x lesen
2. x um 1 erhöhen
3. x schreiben

Das kommt uns bekannt vor...

Speicherkonsistenzfehler lösen

* Lösungsversuch 2: In die Java-Doku schauen und "Synchronisierte Methoden" finden.
* Durch Verwendung des Schlüsselworts **synchronized** signalisieren wir der JVM:

"Diese Methode muss atomar ausgeführt werden - ein Thread darf bei der Ausführung dieser Methode nicht unterbrochen werden."

https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Speicherkonsistenzfehler - Codebeispiel

```
1 public class MemoryConsistency1 {
2     static int x = 0;
3
4     static synchronized int incX(){
5         return x++;
6     }
7
8     ... more code here, see next slide
9 }
```

Speicherkonsistenzfehler - Codebeispiel

```
1 public static void main(String[] args) {
2   new Counter().start();
3   new Counter().start();
4   new Counter().start();
5   new Counter().start();
6 }
```

Läuft!

```
1 Thread-1 wrote: 1
2 Thread-3 wrote: 3
3 Thread-2 wrote: 2
4 Thread-0 wrote: 0
5 Thread-0 wrote: 7
6 Thread-2 wrote: 6
7 Thread-1 wrote: 4
8 Thread-3 wrote: 5
9 Thread-0 wrote: 10
10 Thread-1 wrote: 11
11 ...
```

Coding Time!

Threads - Codingaufgabe 1

30m

* Schreibt eine Klasse "Alphabet", die als Attribut einen leeren String enthält (kann eure Main-Klasse sein).
* Baut mehrere "AlphabetBuilder" als Runnables oder Subklassen von Thread, die:
  1. den letzten Buchstaben des Strings lesen
  2. den nächsten Buchstaben dranhängen, dann 200ms schlafen (wenn der String noch leer ist, soll ein "A" angehängt werden)
  3. sich beenden, sobald das keine "z" erreicht ist.

Threads - Codingaufgabe 1

30m

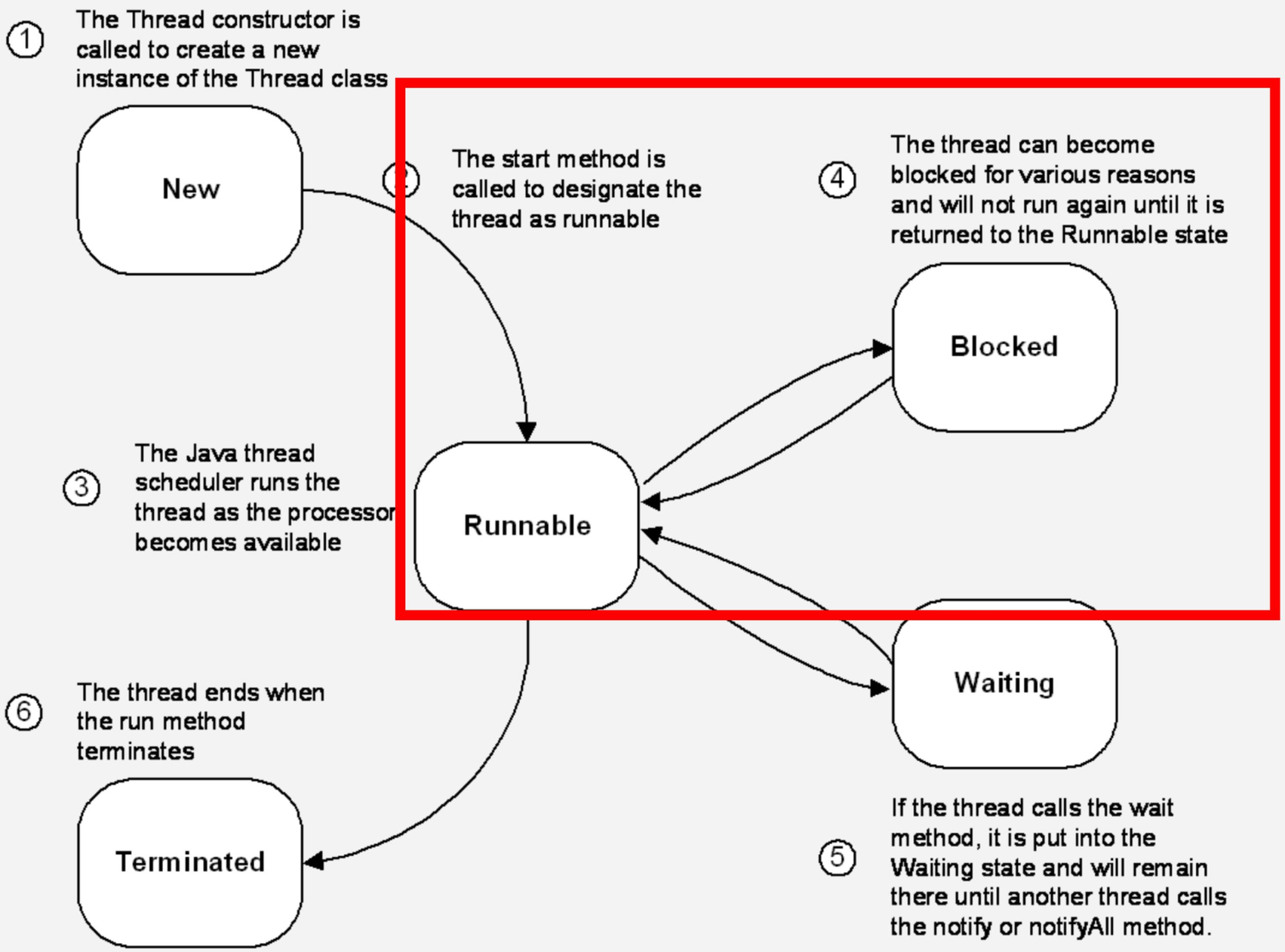
* Gebt nach jedem Anhängen eines Zeichens den String aus.
* Was fällt euch auf?
* Wie lässt sich das Problem lösen?

Hinweise:

Nutzt den Datentyp char - er lässt sich inkrementieren, um das nächste Zeichen zu erhalten!

Nutzt String.length() und String.charAt()

Interrupts



http://voyager.deanza.edu/~hso/cis35a/lecture/java14/intro/lifecycle.html

Blockieren durch sleep()

* Durch Thread.sleep() kann ein Thread von sich aus pausieren

```
1 Runnable task1 = new Runnable() {
2     public void run() {
3     for (int i = 10; i >= 0; i--) {
4       System.out.println(i);
5       try {
6         Thread.sleep(500);
7       } catch (InterruptedException ex) {
8         // Ignore
9       }
10    }
11  }
12 };
13
14 Thread t1 = new Thread(task1);
```

Interrupts

* Ein Interrupt ist eine Nachricht von außen an ein Thread
* Üblich: Ein Thread wird dadurch zur Terminierung aufgefordert
* Thread definiert selbst, ob und wie es auf einen Interrupt reagiert!
* Zwei Möglichkeiten:
  1. Aufruf von Methoden, die eine InterruptedException werfen
  2. Manueller Aufruf von Thread.interrupted()

Interrupts

* Jedes Mal, wenn Thread.sleep() aufgerufen wird, geht der Thread kurz in den Zustand blocked und kann dort interrupted werden.

```
1 public void run() {
2   while(true) {
3     System.out.println("Hello!");
4     try {
5       Thread.sleep(100);
6     } catch (InterruptedException e) {
7       // We've been interrupted! Let's stop it.
8       return;
9     }
10  }
11 }
```

Interrupts

* Wenn ein Thread nicht ständig schlafen will, kann er auch manuell prüfen, ob er interrupted wurde.

```
1 for (int i = 0; i < inputs.length; i++) {
2     performAction(inputs[i]);
3     if (Thread.interrupted()) {
4         // We've been interrupted: no more Action.
5         return;
6     }
7 }
```

Interrupts - Blocked vs Running / Ready

* Nochmal zur Klarstellung:
* Ein Thread wird "Blocked", wenn er diesen Zustand manuell herbeiführt (zB mit sleep()) oder auf eine andere, blockierende Ressource wartet (zB Datenbank, Festplatte)
* Ein Thread kann - unabhängig davon - jederzeit mitten in seiner Ausführung durch das Betriebssystem von "Running" in "Ready" gesetzt werden, wodurch ein anderer Thread die Kontrolle übernimmt.

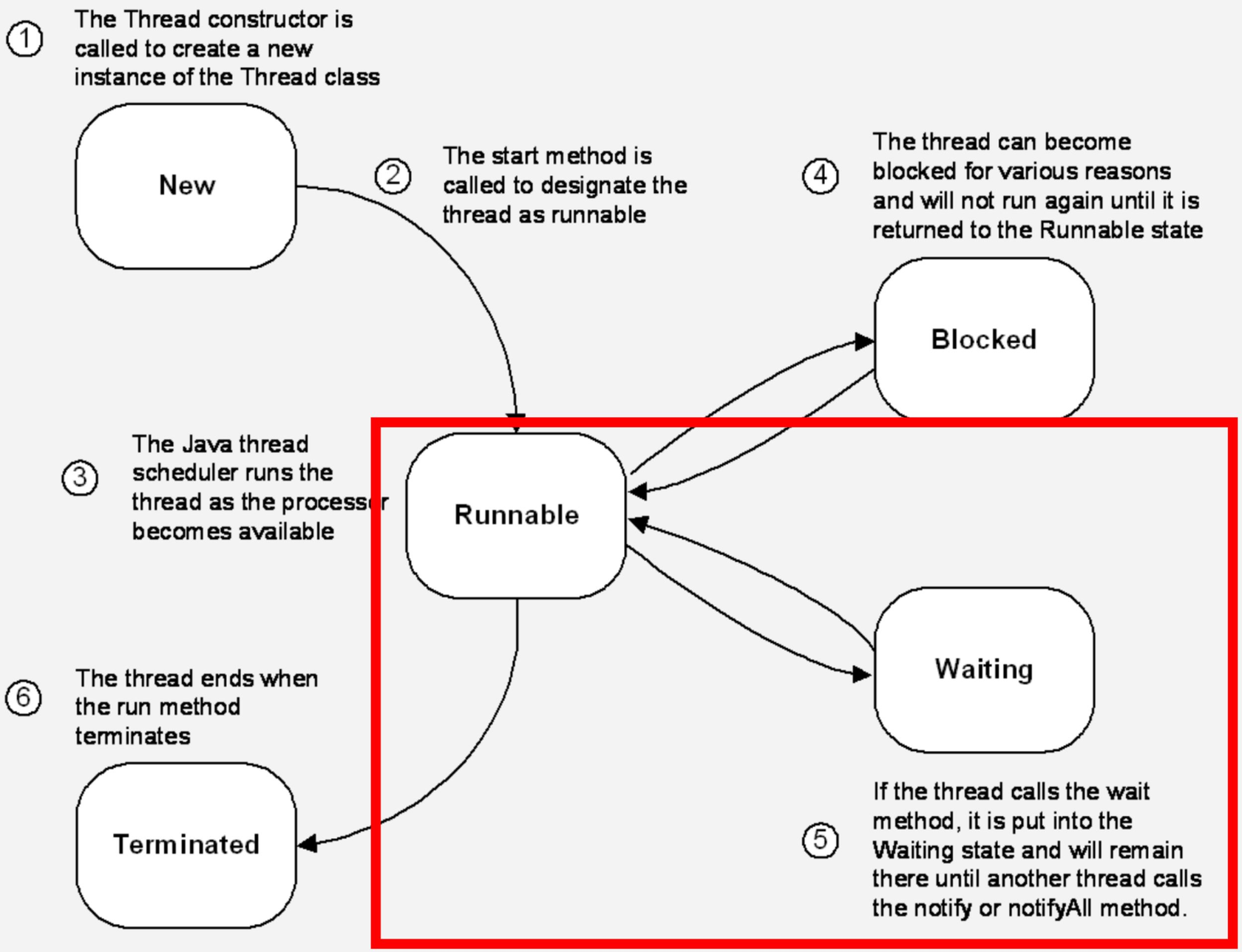
Coding Time!

Threads - Codingaufgabe 2

30m

* Schreibt ein Programm, welches einen zweiten Thread (zusätzlich zum Main-Thread) startet, der mehrere Nachrichten ausgibt und dazwischen jeweils eine Sekunde wartet.
* Der Main-Thread startet die Nachrichten-Queue und wartet dann zufällig zwischen 1 und 5 Sekunden. (Math.random())
* Ist der zweite Thread dann noch nicht beendet, wird er interrupted. (prüft Thread.isAlive())
* Beim Interrupten sollte der zweite Thread eine Nachricht ausgeben ("I wasn't done yet!")

Kommunikation zwischen Threads

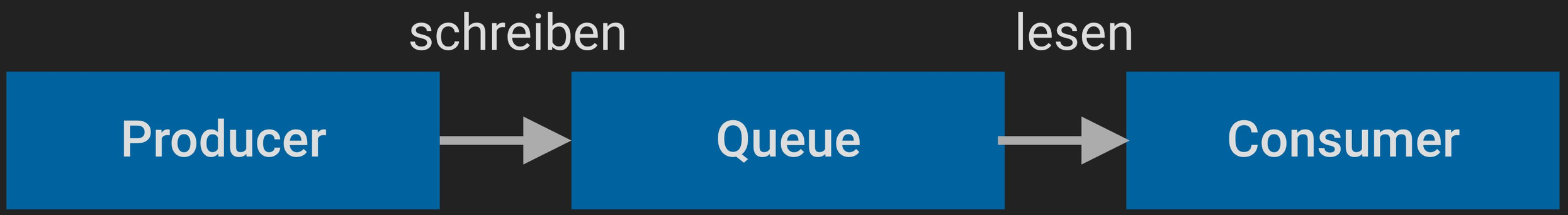


http://voyager.deanza.edu/~hso/cis35a/lecture/java14/intro/lifecycle.html

Anwendungsfall: Consumer-Producer-Problem

* Zwei Threads arbeiten im Wechsel auf dem gleichen Datenobjekt
* Producer ändert den Wert und damit den Zustand des Objektes
* Consumer wartet jeweils auf Änderungen des Producers
* Consumer liest den geänderten Wert aus
* Producer wartet, bis der Consumer seinen Teil abgearbeitet hat

Anwendungsfall: Consumer-Producer-Problem



Anwendungsfall: Consumer-Producer-Problem

* Anwendungsbeispiele für das Consumer-Producer-Problem:
  * Warteschlangen mit einer bestimmten Kapazität
  * Beschaffung und Verarbeitung von Daten
  * Lese-Schreibe-Problematiken
  * Webserver, der Requests zahlreicher Browser entgegen nimmt und verarbeitet

wait(), notify() und notifyAll()

`Object.wait()`

Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

`Object.notify()`

Wakes up a single thread that is waiting on this object's monitor.

`Object.notifyAll()`

Wakes up all threads that are waiting on this object's monitor.

wait(), notify() und notifyAll()

* wait() versetzt einen Thread in einen Wartezustand
* Der Wartezustand bezieht sich immer auf ein bestimmtes Objekt - man sagt, ein Thread wartet an einem Objekt
* Ein Objekt, an dem gewartet wird, heißt **Monitor**
* In der Java-Doku ist oft die Rede vom Monitor eines Objektes (an Object's Monitor) - damit ist ebenfalls das Objekt gemeint, an welchem gewartet wird
* Gewartet werden kann nur innerhalb von **synchronized** Blöcken

https://de.wikipedia.org/wiki/Monitor_(Informatik)

wait(), notify() und notifyAll()

* Solange Threads warten, befinden sie sich im Zustand "Waiting"
* Während ein Thread wartet, kann es Interrupts empfangen - deshalb kann wait() eine InterruptedException werfen
* notify() bzw. notifyAll() wecken ein bzw. alle Threads, die an einem Monitor warten
* Deren Ausführung wird dann direkt hinter dem Methodenaufruf von wait() fortgesetzt
* Mit wait(long timeout) kann ein maximaler Wartezeitraum (in ms) angegeben werden

Consumer-Producer-Problem: Codebeispiel

```
1 public class Lager {
2   private int bestand = 5;
3   private final int maxBestand = 10;
4
5   ...
6 }
```

Consumer-Producer-Problem: Codebeispiel

```
1 public synchronized void addBestand() throws InterruptedException {
2   if (bestand == maxBestand) {
3     // Lager voll - warten bis wir geweckt werden
4     wait();
5   }
6   bestand++;
7   if (bestand > 8) {
8     // Lager fast voll - ggf. wartende Consumer wecken
9     notifyAll();
10  }
11 }
```

Consumer-Producer-Problem: Codebeispiel

```
1 public synchronized void removeBestand() throws InterruptedException {
2   if (bestand == 0) {
3     // Lager leer - warten bis wir geweckt werden
4     wait();
5   }
6   bestand--;
7   if (bestand < 2) {
8     // Lager fast leer - ggf. wartende Producer wecken
9     notifyAll();
10  }
11 }
```

Consumer-Producer-Problem: Codebeispiel

```
1 public class Producer implements Runnable {
2   private Lager lager;
3   public Producer(Lager lager) { this.lager = lager; }
4
5   public void run() {
6     while (true) {
7       try {
8         lager.addBestand();
9         Thread.sleep(200);
10      } catch (InterruptedException ex) {
11        //
12      }
13    }
14  }
15 }
```

Consumer-Producer-Problem: Codebeispiel

```
1  public class Consumer implements Runnable {
2    private Lager lager;
3    public Consumer(Lager lager) { this.lager = lager; }
4
5    public void run() {
6      while (true) {
7        try {
8          lager.removeBestand();
9          Thread.sleep(2000);
10       } catch (InterruptedException ex) {
11         //
12       }
13     }
14   }
15 }
```

Consumer-Producer-Problem: Codebeispiel

```
1 public class Main {
2   public static void main(String[] args) {
3     Lager lager = new Lager();
4     Consumer c = new Consumer(lager);
5     Producer p = new Producer(lager);
6
7     new Thread(c).start();
8     new Thread(p).start();
9   }
10 }
```

Consumer-Producer-Problem: Skalierung

* Mit dem Consumer-Producer-Pattern lassen sich Anwendungen besser skalieren
* Beispiel: Mein Consumer ist sehr langsam - also starte ich einfach noch zwei weitere!
* So lassen sich Anwendungen entsprechend der Last skalieren
* Services wie AWS machen das automatisch (Autoscaling)

Consumer-Producer-Problem: Skalierung

```
1 public class Main {
2   public static void main(String[] args) {
3     Lager lager = new Lager();
4     Consumer c = new Consumer(lager);
5     Producer p = new Producer(lager);
6
7     // Unsere Consumer sind langsam, also brauchen wir mehr!
8     new Thread(c).start();
9     new Thread(c).start();
10    new Thread(c).start();
11    new Thread(p).start();
12  }
13 }
```