TECHNISCHE UNIVERSITÄT DARMSTADT

Kapitel 09: Threads

Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

Verschachtelte Klassen (nested classes)

Wir brauchen eine kleine Vorarbeit, bevor wir zum eigentlichen Thema des Kapitels kommen können.

Klassen kann man auch innerhalb anderer Klassen definieren. Der Fachbegriff dazu lautet *verschachtelte Klassen*, englisch *nested classes*, was deutsch eher *eingebettete Klassen* bedeutet. Begrifflicher Zusammenhang: Dass zwei Klassen verschachtelt sind, bedeutet, dass die eine in der anderen eingebettet ist.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int n;
   private Y a;
   public X () {
      a = new Y ();
   }
   public void m1 () {
      a.m2 ();
   }
   private class Y {
      public void m2 () {
         n = 123;
      }
   }
}
```

Ein kleines illustratives Beispiel zur Erläuterung.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
    private int n;
    private Y a;
    public X () {
        a = new Y ();
    }
    public void m1 () {
        a.m2 ();
    }
    private class Y {
        public void m2 () {
            n = 123;
        }
    }
}
```

Alles, was Sie auf dieser Folie sehen, gehört zur Klasse X.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int n;
   private Y a;
   public X () {
      a = new Y ();
   }
   public void m1 () {
      a.m2 ();
   }
   private class Y {
      public void m2 () {
         n = 123;
      }
   }
}
```

Auch diese Definition einer Klasse Y gehört also zu X.

Wir sagen, X und Y sind *verschachtelt*, oder auch, Y ist in X eingebettet. Wir nennen X auch die *äußere* und Y die *innere* Klasse.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int n;
   private Y a;
   public X () {
      a = new Y ();
   }
   public void m1 () {
      a.m2 ();
   }
   private class Y {
      public void m2 () {
         n = 123;
      }
   }
}
```

Eine Klasse wie X kann weiterhin nur entweder mit oder ohne public definiert werden, kein private oder protected ist möglich.

Eine innere Klasse wie Y kann hingegen wie ein Attribut oder eine Methode entweder public oder private oder protected sein. Die Bedeutung ist dieselbe wie bei Attributen und Methoden.

***Erinnerung*: Maximal *eine* Klasse kann in einer Java-Quelldatei public sein, und die Datei muss dann den Namen dieser Klasse haben (mit Endung .java).**

***Erinnerung*: public bedeutet, man kann von überall her darauf zugreifen, mit private geht das nur von der Klasse selbst aus, ohne Schlüsselwort von überall im selben Package her und mit protected von zwei Bereichen aus: überall im selben Package her und von der abgeleiteten Klasse aus.**

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int n;
   private Y a;
   public X () {
      a = new Y ();
   }
   public void m1 () {
      a.m2 ();
   }
   private class Y {
      public void m2 () {
         n = 123;
      }
   }
}
```

So wie bei einer beliebigen anderen Klasse, kann in X ein Attribut von Y eingerichtet, ein Objekt mit new erzeugt und auf Attribute und Methoden, die public sind, zugegriffen werden.

Dass die Klasse Y selbst private ist, ist dafür natürlich kein Hinderungsgrund, denn Y gehört ja zu X. Aber außerhalb von X kann auf Y wegen private nicht zugegriffen werden.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  private int n;
  private Y a;
  public X () {
    a = new Y ();
  }
  public void m1 () {
    a.m2 ();
  }
  private class Y {
    public void m2 () {
      n = 123;
    }
  }
}
```

Umgekehrt kann in Y auch auf Attribute und Methoden von X zugegriffen werden, die private definiert sind.

Der hier gezeigte Zugriff auf ein *Objekt*attribut wirft allerdings die Frage auf, von welchem Objekt das n genommen wird.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int n;
   private Y a;
   public X () {
      a = new Y ();
   }
   public void m1 () {
      a.m2 ();
   }
   private class Y {
      public void m2 () {
         n = 123;
      }
   }
}
```

Die Antwort ist einfach: Nur eine Objektmethode von X kann ein Objekt von Y so wie hier im Konstruktor einrichten. Eine Objektmethode wird immer mit einem Objekt aufgerufen. Dieses Objekt ist dann das, auf dessen Attribut zugegriffen wird.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int n;
   private Y a;
   public X () {
      a = new Y ();
   }
   public void m1 () {
      a.m2 ();
   }
   public class Y {
      public void m2 () {
         n = 123;
      }
   }
}
```

Wenn die innere Klasse nicht private wie bisher, sondern public ist, kann auf sie auch außerhalb von X zugegriffen werden.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {

   .........

    public class Y {
      public void m2 () {
         n = 123;
      }
   }
}
```

```
X a = new X ();
X.Y b = a.new Y ();
b.m2 ();
X.Y c = a.new Y ();
c.m2 ();
```

So kann man im Falle von public die innere Klasse außerhalb der äußeren Klasse ansprechen: der Name der äußeren Klasse plus der Name der inneren Klasse, durch Punkt getrennt.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

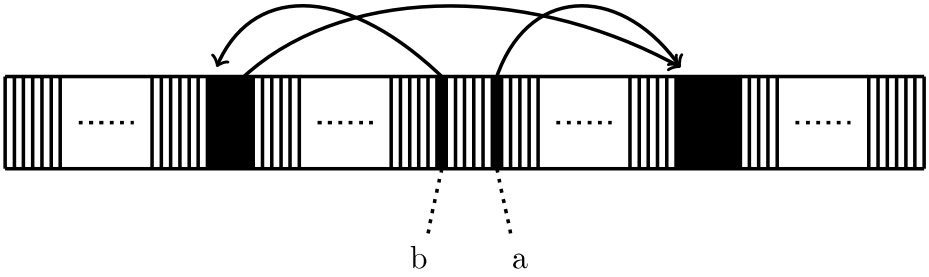
```
public class X {
   private int n;
   private Y y;
   public X () {
      y = new Y ();
   }
   public void m1 () {
      y.m2 ();
   }
```

```
X a = new X ();
X.Y b = a.new Y ();
b.m2 ();
X.Y c = a.new Y ();
c.m2 ();
```

Die Einrichtung eines Objektes von X.Y sieht jetzt leicht anders aus als gewohnt: new wird so verwendet, als würde new zu X gehören. Das ist auch wichtig, denn wie wir gleich sehen werden, braucht ein Objekt von Y ein Objekt von X, auf das es sich bezieht.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT




```
X a = new X ();
X.Y b = a.new Y ();
```

So kann man sich diese Beziehung dann im Computerspeicher vorstellen, unten sehen Sie noch einmal die entscheidenden Zeilen von der letzten Folie. Ein Objekt von Klasse X.Y hat automatisch einen anonymen Verweis auf dasjenige Objekt von Klasse X, mit dem es via a.new eingerichtet wurde. Darüber werden die Zugriffe realisiert, die wir uns als nächstes ansehen werden.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int n;
   private Y y;
   public X () {
      y = new Y ();
   }
   public void m1 () {
      y.m2 ();
   }
```

```
X a = new X ();
X.Y b = a.new Y ();
b.m2 ();
X.Y c = a.new Y ();
c.m2 ();
```

Die Objektmethode m2 von Klasse Y wird mit dem neu eingerichteten Objekt von Klasse Y aufgerufen und greift auf das Objektattribut n von X zu. Wie wir soeben bildlich gesehen haben, ist in dem Objekt von Y, auf das b beziehungsweise c beziehungsweise y verweist, jeweils ein anonymer Verweis auf das Objekt von X, mit dem das jeweilige Objekt von Y eingerichtet wurde.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int n;
   private Y y;
   public X () {
      y = new Y ();
   }
   public void m1 () {
      y.m2 ();
   }
```

```
X a = new X ();
X.Y b = a.new Y ();
b.m2 ();
X.Y c = a.new Y ();
c.m2 ();
```

Wie üblich, werden mit zweimaliger Anwendung von Operator new auch hier wieder zwei verschiedene Objekte erzeugt. Aber beide Objekte von Klasse Y werden mit demselben Objekt von Klasse X erzeugt und verweisen daher beide auf dieses Objekt.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int n;
   private Y y;
   public X () {
      y = new Y ();
   }
   public void m1 () {
      y.m2 ();
   }
```

```
X a = new X ();
X.Y b = a.new Y ();
b.m2 ();
X.Y c = a.new Y ();
c.m2 ();
```

Durch die Verknüpfung mit a in a.new auf der rechten Seite wird dem Compiler gesagt, dass er den anonymen Verweis auf das Objekt von X setzen soll, auf das a verweist. Darüber wird dann das Attribut n in diesem Objekt von X angesprochen. Analog wird dem Compiler links beim Aufruf von new Y gesagt, dass er den anonymen Verweis auf this setzen soll.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int n;
   private Y y;
   public X () {
      y = new Y ();
   }
   public void m1 () {
      y.m2 ();
   }
```

```
X a = new X ();
X.Y b = a.new Y ();
b.m2 ();
X.Y c = a.new Y ();
c.m2 ();
```

Daher wird zwar die Methode m2 mit drei verschiedenen Objekten von Y aufgerufen, aber es wird jedes Mal dieselbe Speicherstelle mit dem Wert 123 überschrieben, nämlich das Attribut n des Objekts von X, das alle drei Objekte von Y gemeinsam haben.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int n1;
   private static int n2;
   private static class Y {
      public void m2 () {
         n1 = 123;
         n2 = 321;
      }
   }
}
```

Wie bei Attributen und Methoden, so gibt es auch bei verschachtelten Klassen eine static-Variante. Dazu wird das Schlüsselwort static vor das Schlüsselwort class gechrieben.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int n1;
   private static int n2;
   private static class Y {
      public void m2 () {
         n1 = 123;
         n2 = 321;
      }
   }
}
```

Analog zu static-definierten Methoden kann auch eine static-definierte innere Klasse nicht auf Objektattribute und Objektmethoden zugreifen, sondern nur auf Klassenattribute und Klassenmethoden.

Verschachtelte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public static class Y {
        public void m () { ..... }
}

X.Y a = new X.Y ();
a.m ();
```

Analog zu static-Attributen und static-Methoden reicht auch bei static-Klassen die Angabe des Klassennamens X; kein Objekt von X ist nötig.

TECHNISCHE UNIVERSITÄT DARMSTADT

Runnable und Thread

Jetzt kommen wir zum Hauptthema des Kapitels.

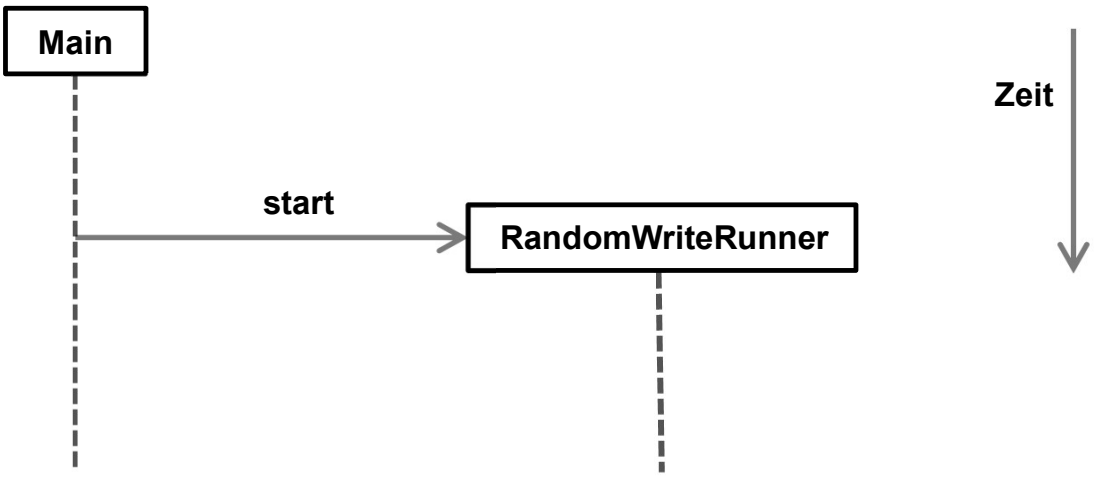
Das Interface Runnable und die Klasse Thread aus dem Package java.lang sind die Grundlage dieses Kapitels. Sie sind ein typisches Beispiel für das allgemeine Programmierprinzip Separation of Concerns: Thread organisiert einen parallel laufenden Prozess, und Runnable enthält dessen Inhalt. Die Art von parallelen Prozessen, die wir uns hier anschauen, heißen *Threads*.

Generell unterscheidet man zwischen echten Prozessen und Threads und nennt Threads häufig leichtgewichtige Prozesse, englisch light-weight process. Echte, schwergewichtige Prozesse, englisch heavy-weight, werden durch das Betriebssystem verwaltet. Dazu gehören typischerweise Webbrowser, Softwareentwicklungsumgebungen, Kommunikationsprogramme und vieles mehr. Auch die Ausführung eines Java-Programms ist in diesem Sinne ein echter Prozess.

Siehe Zusammenfassung Programme und Prozesse.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

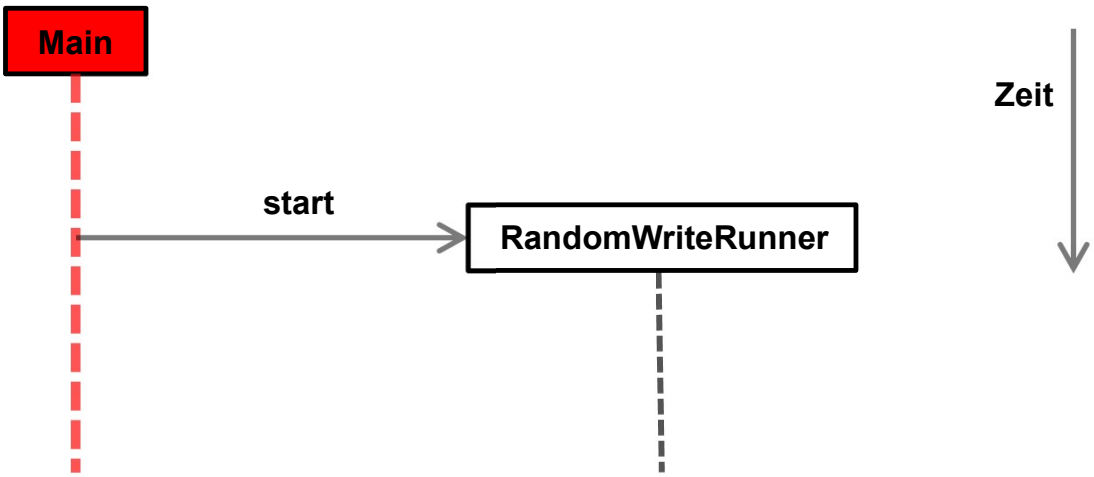


Als erstes sehen wir uns das Gesamtbild an, das am Ende herauskommen soll. Dazu werden wir die hier bezeichneten Klassen Main und RandomWriteRunner selbst zu implementieren haben; alles andere kommt aus der Java-Standardbibliothek.

was Sie hier sehen, ist eine Art von UML-Diagramm, die wir bisher noch nicht gesehen haben: ein Sequenzdiagramm.

Runnable und Thread

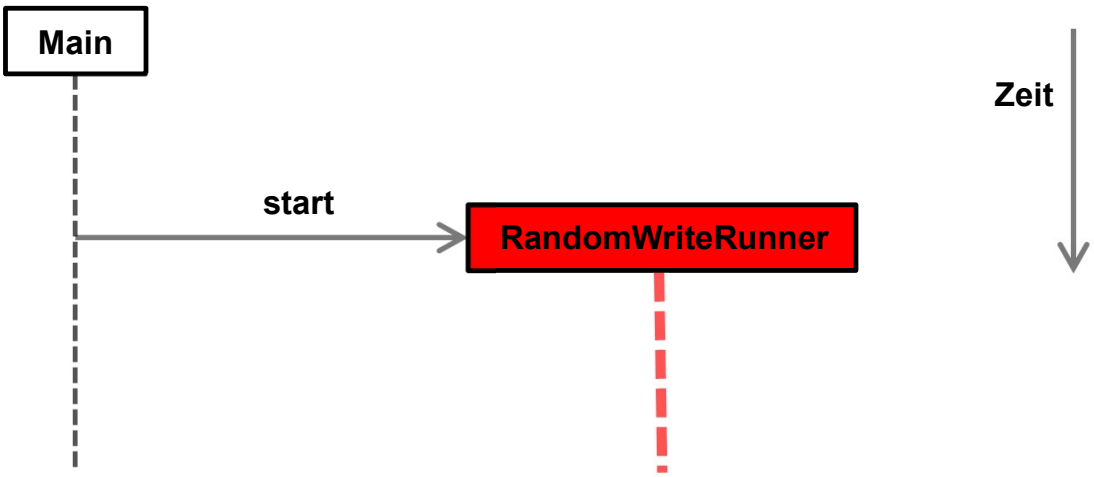
TECHNISCHE UNIVERSITÄT DARMSTADT



Der rot gezeichnete Teil ist das, was wir bisher ohne Threads gemacht haben: Eine main-Methode einer Klasse wird gestartet, und die Abarbeitung des Programms besteht darin, dass diese main-Methode abgearbeitet wird und dabei diverse andere Klassen angesprochen werden. Die Klasse mit der main-Methode heißt hier einfach Main, deshalb steht Main oben. Die gestrichelte vertikale Linie symbolisiert den Ablauf des Programms in der Zeit.

Runnable und Thread

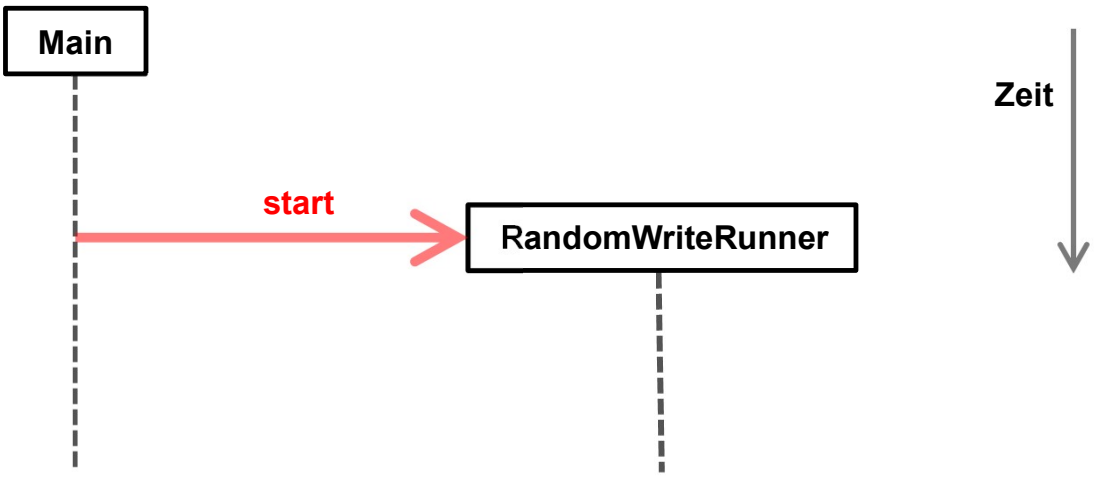
TECHNISCHE UNIVERSITÄT DARMSTADT



Das ist jetzt neu: ein zweiter Thread. Dieser wird über eine Klasse namens RandomWriteRunner gestartet, die wir wie gesagt noch zu implementieren haben.

Runnable und Thread

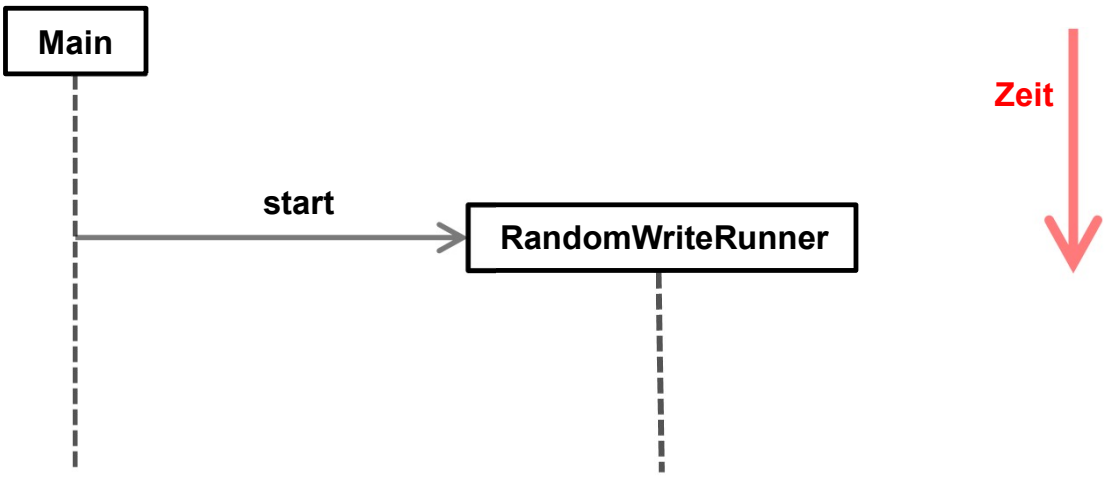
TECHNISCHE UNIVERSITÄT DARMSTADT



Diesen zweiten Thread starten wir aus dem ersten Thread heraus. Threads müssen nicht unabhängig voneinander nebeneinander herlaufen, sondern können sich auch Nachrichten schicken. Der Pfeil wird mit der Art der Nachricht annotiert.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT



Eine wichtige Anmerkung zur zeitlichen Darstellung: Die Zeitachse ist keinesfalls als numerische Skala zu sehen, denn wann etwas Bestimmtes wie zum Beispiel das Senden einer Nachricht passiert, hängt von unzähligen Unwägbarkeiten ab. Nur die *Reihenfolge* einzelner Aktionen und Zustandsänderungen kann man ablesen, ihre vertikale Höhe hat nichts zu bedeuten.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public static void main ( String[ ] args ) {
    Predicate<Integer> pred = new .....;
    OutputStream out = new .....;
    Runnable runnable = new RandomWriteRunner ( pred, out );
    new Thread(runnable).start();
    doTheOtherThings();
  }
}
```

Kommen wir zuerst zur Klasse Main. Wie gesagt, ist das eine Klasse, wie wir sie schon öfters implementiert haben, nur dass sie ein paar neue Bestandteile enthält.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public static void main ( String[ ] args ) {
    Predicate<Integer> pred = new .....;
    OutputStream out = new .....;
    Runnable runnable = new RandomWriteRunner ( pred, out );
    new Thread(runnable).start();
    doTheOtherThings();
  }
}
```

Erinnerung: Methode main ist die Einstiegsmethode für diverse Java-Ausführungsprogramme. Der Kopf der Methode muss aber exakt so wie hier aussehen: Sie muss void und eine public-Klassenmethode sein und hat einen einzelnen Parameter vom Typ String-Array.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public static void main ( String[ ] args ) {
    Predicate<Integer> pred = new .....;
    OutputStream out = new .....;
    Runnable runnable = new RandomWriteRunner ( pred, out );
    new Thread(runnable).start();
    doTheOtherThings();
  }
}
```

Für RandomWriteRunner werden wir ein Prädikat auf Integer sowie eine Datensenke benötigen: RandomWriteRunner soll Zufallszahlen auf die Datensenke schreiben, bis die Anzahl der generierten Zufallszahlen das Prädikat erfüllt. Was für ein Prädikat und was für eine Datensenke dafür genau gewählt werden, ist für uns hier irrelevant.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public static void main ( String[ ] args ) {
    Predicate<Integer> pred = new .....;
    OutputStream out = new .....;
    Runnable runnable = new RandomWriteRunner ( pred, out );
    new Thread(runnable).start();
    doTheOtherThings();
  }
}
```

Runnable ist ein Interface im Paket java.lang. Die noch zu schreibende Klasse RandomWriteRunner implementiert dieses Interface.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public static void main ( String[ ] args ) {
    Predicate<Integer> pred = new .....;
    OutputStream out = new .....;
    Runnable runnable = new RandomWriteRunner ( pred, out );
    new Thread(runnable).start();
    doTheOtherThings();
  }
}
```

Klasse Thread in Package java.lang wird mit einem Runnable eingerichtet. Das ist die Verknüpfung zwischen der Funktionalität des Threads, die in Runnable implementiert wird, und der Organisation von Threads allgemein, die von Klasse Thread realisiert wird.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public static void main ( String[ ] args ) {
    Predicate<Integer> pred = new .....;
    OutputStream out = new .....;
    Runnable runnable = new RandomWriteRunner ( pred, out );
    new Thread(runnable).start();
    doTheOtherThings();
  }
}
```

Nachdem wir den Thread erzeugt haben, wird Main ihm keine weiteren Nachrichten mehr schicken und auch keine Nachrichten von dort erwarten; der zweite Thread läuft in diesem ersten einfachen Beispiel alleine vor sich hin, ohne weitere Interaktion mit Main. Daher weisen wir die Adresse des von Operator new erzeugten und zurückgelieferten Objektes ausnahmsweise einmal *nicht* einer Referenz zu, sondern vergessen sie einfach, denn wir brauchen sie nicht.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public static void main ( String[ ] args ) {
    Predicate<Integer> pred = new .....;
    OutputStream out = new .....;
    Runnable runnable = new RandomWriteRunner ( pred, out );
    new Thread(runnable).start();
    doTheOtherThings();
  }
}
```

das ist die Realisierung der Nachricht „start“: Methode start wird hier auf das neu erzeugte Thread-Objekt angewendet. Das startet den eigentlichen Thread. Von diesem Moment an besteht die Programmausführung aus *zwei* Threads: dem Thread, in dem Methode main von Klasse Main gerade abläuft, und dem neuen Thread.

Das, was wir bisher immer als Programmausführung oder ähnlich bezeichnet haben, ist also in Wirklichkeit die Ausführung eines einzelnen Threads im Programm. Solange wir keine neuen Threads gestartet haben, konnte uns der Unterschied egal sein, da ja der ganze Prozess mit diesem einen Thread identisch war. Ab jetzt müssen wir aber zwischen Programmausführung und einzelnen Threads *in* der Programmausführung unterscheiden.

***Nebenbemerkung*: Startet man einen Thread aus Versehen ein zweites Mal, wird eine Exception vom Typ IllegalThreadStateException geworfen. Diese Exception-Klasse ist aber indirekt von RuntimeException abgeleitet, so dass sie nicht gefangen werden muss. In unseren einfachen. überschaubaren Beispielen können wir problemlos sicher sein, dass kein Thread mehr als einmal gestartet**

wird.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public static void main ( String[ ] args ) {
    Predicate<Integer> pred = new .....;
    OutputStream out = new .....;
    Runnable runnable = new RandomWriteRunner ( pred, out );
    new Thread(runnable).start();
    doTheOtherThings();
  }
}
```

Der entscheidende Punkt ist jetzt: Beide Threads laufen einfach parallel nebeneinander her. Der neu gestartete Thread erzeugt Zufallszahlen und schreibt sie auf seinen PrintStream, bis die Abbruchbedingung erreicht ist, dann beendet er sich. Parallel dazu arbeitet der ursprüngliche Thread einfach weiter, so als würde es den anderen Thread gar nicht geben. Beide Threads laufen in der Tat völlig unbeeinflusst nebeneinander her. Ist der eine von beiden zu Ende, läuft der andere einfach weiter, bis er ebenfalls seine Arbeit beendet hat.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public static void main ( String[ ] args ) {
    Predicate<Integer> pred = new .....;
    OutputStream out = new .....;
    Runnable runnable = new RandomWriteRunner ( pred, out );
    new Thread(runnable).start();
    doTheOtherThings();
  }
}
```

***Erinnerung*: Der Begriff „parallel“ ist allerdings interpretationsbedürftig, siehe Abschnitt „Allgemein: Programmablauf“ in Kapitel 01a. Auf einer Hardwareplattform mit nur einem Prozessor werden die einzelnen Prozesse und die Threads, aus denen sich jeder Prozess zusammensetzt, wechselweise ausgeführt. Die Wechsel passieren so schnell, dass der Mensch meist nichts davon mitbekommt und den subjektiven Eindruck hat, die Prozesse und Threads würden wirklich parallel laufen. Bei mehreren Prozessoren werden die Threads potentiell auf die Prozessoren verteilt. Zwei Threads auf unterschiedlichen Prozessoren laufen real parallel.**

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class RandomWriteRunner implements Runnable {
  ..........

  public void run() {
     while ( ! stopCriterion.test ( generatedSoFar ) ) {
         out.println ( random.nextDouble() );
         generatedSoFar++;
     }
  }
}
```

Nun fehlt noch die Klasse RandomWriteRunner, in der implementiert ist, was der zweite Thread macht.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class RandomWriteRunner implements Runnable {
  .........

  public void run() {
     while ( ! stopCriterion.test ( generatedSoFar ) ) {
        out.println ( random.nextDouble() );
        generatedSoFar++;
     }
  }
}
```

Wie gesagt, ist der Inhalt eines Threads in eine Klasse zu schreiben, die Runnable implementiert.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class RandomWriteRunner implements Runnable {
  .........

  public void run() {
     while ( ! stopCriterion.test ( generatedSoFar ) ) {
        out.println ( random.nextDouble() );
        generatedSoFar++;
     }
  }
}
```

Runnable ist ein Functional Interface. Die funktionale Methode heißt run, hat keine Parameter und ist void.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class RandomWriteRunner implements Runnable {
  .........

  public void run() {
    while ( ! stopCriterion.test ( generatedSoFar ) ) {
        out.println ( random.nextDouble() );
        generatedSoFar++;
    }
  }
}
```

Der Inhalt von Methode run ist das, was der Thread machen soll: Solange eine bestimmte Abbruchbedingung nicht erfüllt ist, sollen zufällige double-Werte irgendwohin geschrieben werden.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class RandomWriteRunner implements Runnable {
  .........

  public void run() {
     while ( ! stopCriterion.test ( generatedSoFar ) ) {
        out.println ( random.nextDouble() );
        generatedSoFar++;
     }
  }
}
```

Die Abbruchbedingung bekommt die Anzahl der bisher generierten Zufallszahlen als Parameterwert. Wir werden gleich zwei Implementationen der Abbruchbedingung sehen; die eine verwendet diesen Parameterwert, die andere ignoriert ihn.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class RandomWriteRunner implements Runnable {
  .....

  public void run() {
     while ( ! stopCriterion.test ( generatedSoFar ) ) {
        out.println ( random.nextDouble() );
        generatedSoFar++;
     }
  }
}
```

Das erfordert natürlich ein paar Vorarbeiten in Form von Attributen der Klasse RandomWriteRunner und natürlich auch einen Konstruktor. Das sehen wir uns als nächstes an.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class RandomWriteRunner implements Runnable {
  private Random random;
  private Predicate<Integer> stopCriterion;
  private int generatedSoFar;
  private PrintStream out;
  public RandomWriteRunner ( Predicate<Integer> stopCriterion, PrintStream out ) {
    random = new Random();
    this.stopCriterion = stopCriterion;
    generatedSoFar = 0;
    this.out = out;
  }
  ..........
}
```

Das, was hier unten mit Punkten angedeutet wird, haben wir uns soeben angesehen.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class RandomWriteRunner implements Runnable {
  private Random random;
  private Predicate<Integer> stopCriterion;
  private int generatedSoFar;
  private PrintStream out;
  public RandomWriteRunner ( Predicate<Integer> stopCriterion, PrintStream out ) {
    random = new Random();
    this.stopCriterion = stopCriterion;
    generatedSoFar = 0;
    this.out = out;
  }
  .........
}
```

Das sind die vier Attribute, die in Methode run verwendet wurden.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class RandomWriteRunner implements Runnable {
  private Random random;
  private Predicate<Integer> stopCriterion;
  private int generatedSoFar;
  private PrintStream out;
  public RandomWriteRunner ( Predicate<Integer> stopCriterion, PrintStream out ) {
    random = new Random();
    this.stopCriterion = stopCriterion;
    generatedSoFar = 0;
    this.out = out;
  }
  .........
}
```

***Erinnerung*: Klasse Random aus Package java.util. bietet die Generierung von beliebig vielen Zufallszahlen aus verschiedenen primitiven Datentypen.**

In der Methode run haben wir die Methode nextDouble von Klasse Random verwendet, die bei jedem Aufruf eine neue Zufallszahl vom Typ double generiert und zurückliefert.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class RandomWriteRunner implements Runnable {
  private Random random;
  private Predicate<Integer> stopCriterion;
  private int generatedSoFar;
  private PrintStream out;
  public RandomWriteRunner ( Predicate<Integer> stopCriterion, PrintStream out ) {
    random = new Random();
    this.stopCriterion = stopCriterion;
    generatedSoFar = 0;
    this.out = out;
  }
  .........
}
```

***Erinnerung*: Das generische Interface Predicate aus Package java.util.function haben wir schon in Kapitel 04c gesehen, Abschnitt über Functional Interfaces und Lambda-Ausdrücke. Die funktionale Methode heißt test, liefert ein boolean zurück und hat einen Parameter vom generischen Typ. Wir hatten einen int-Wert übergeben; der wird durch Boxing zu einem Integer-Objekt, das passt also.**

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class RandomWriteRunner implements Runnable {
  private Random random;
  private Predicate<Integer> stopCriterion;
  private int generatedSoFar;
  private PrintStream out;
  public RandomWriteRunner ( Predicate<Integer> stopCriterion, PrintStream out ) {
    random = new Random();
    this.stopCriterion = stopCriterion;
    generatedSoFar = 0;
    this.out = out;
  }
  .....
}
```

Der PrintStream wird irgendwo anders eingerichtet, so dass Methode run mit jeder beliebigen möglichen Datensenke arbeiten kann.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class CountPredicate implements Predicate<Integer> {
  private int max;
  public CountPredicate ( int max ) {
    this.max = max;
  }
  public boolean test ( Integer current ) {
     return current >= max;
  }
}
```

Wie angekündigt, implementieren wir die Abbruchbedingung zweimal, zuerst die Implementation, die den Parameter verwendet.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class CountPredicate implements Predicate<Integer> {
  private int max;
  public CountPredicate ( int max ) {
    this.max = max;
  }
  public boolean test ( Integer current ) {
     return current >= max;
  }
}
```

Die Idee bei dieser Abbruchbedingung ist, dass eine Maximalzahl von Zufallswerten vorgegeben wird. Sobald diese Anzahl generiert ist, greift das Abbruchkriterium.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class CountPredicate implements Predicate<Integer> {
  private int max;
  public CountPredicate ( int max ) {
    this.max = max;
  }
  public boolean test ( Integer current ) {
     return current >= max;
  }
}
```

Aus dem Kontext von vor ein paar Folien wissen wir, dass current nicht größer als max sein kann. Aber es kann ja sein, dass sich der Kontext einmal ändert oder dass CountPredicate in einem anderen Kontext verwendet wird. Mit test auf größer-gleich anstelle von Gleichheit sind wir da auf der sicheren Seite.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class TimeoutPredicate implements Predicate<Integer> {
  private Calendar latestTime;
  public TimeoutPredicate ( Calendar latestTime ) {
    this.latestTime = latestTime;
  }
  public boolean test ( Integer current ) {
     return latestTime.before ( Calendar.getInstance() );
  }
}
```

Nun eine zweite Implementation der Abbruchbedingung, die, wie gesagt, *nicht* den Parameterwert in Methode test verwendet.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class TimeoutPredicate implements Predicate<Integer> {
  private Calendar latestTime;
  public TimeoutPredicate ( Calendar latestTime ) {
    this.latestTime = latestTime;
  }
  public boolean test ( Integer current ) {
     return latestTime.before ( Calendar.getInstance() );
  }
}
```

***Erinnerung*: Die Klasse Calendar in Package java.util verwaltet kalendarische Daten inklusive Uhrzeit. Die Klassenmethode getInstance liefert einen Verweis auf ein Calendar-Objekt zurück, das exakt den Moment repräsentiert, in dem die Methode aufgerufen wurde.**

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class TimeoutPredicate implements Predicate<Integer> {
  private Calendar latestTime;
  public TimeoutPredicate ( Calendar latestTime ) {
    this.latestTime = latestTime;
  }
  public boolean test ( Integer current ) {
     return latestTime.before ( Calendar.getInstance() );
  }
}
```

Diese Methode ist jetzt neu. Es ist ja ziemlich mühselig, zwei Zeitpunkte miteinander zu vergleichen: Wenn die Jahreszahlen unterschiedlich sind, dann ist *das* das Vergleichskriterium; sind die Jahreszahlen gleich, aber die Monate unterschiedlich, dann entscheidet der Monat; sind Jahr und Monat gleich, der Tag aber nicht, entscheidet der Tag; und so weiter, bis hinunter zur Millisekunde. Diese Mühsal nehmen uns die Methoden before und after ab. Der formale Parametertyp ist zwar Object, aber es ist schon so gedacht, dass der aktuale Parameter wie hier ein Calendar ist. Methode before liefert in diesem Fall genau dann true zurück, wenn das Calendar-Objekt, mit dem before aufgerufen wird, einen Zeitpunkt repräsentiert, der *vor* dem vom Parameter repräsentierten Zeitpunkt liegt.

Runnable und Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class TimeoutPredicate implements Predicate<Integer> {
  private Calendar latestTime;
  public TimeoutPredicate ( Calendar latestTime ) {
    this.latestTime = latestTime;
  }
  public boolean test ( Integer current ) {
     return latestTime.before ( Calendar.getInstance() );
  }
}
```

Die Abbruchbedingung bei Klasse TimeoutPredicate ist also, dass der Zeitpunkt überschritten ist, der im Konstruktor durch ein Calendar-Objekt angegeben ist.

TECHNISCHE UNIVERSITÄT DARMSTADT

Klasse Thread

Im einführenden Beispiel hatten wir uns auf die Funktionalität des Threads und daher auf Interface Runnable konzentriert; Klasse Thread kam nur ganz kurz vor. Bevor es weitergeht mit der Funktionalität von Threads, schauen wir uns zur Abrundung noch ein paar ausgewählte Methoden von Klasse Thread an.

Klasse Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

static currentThread

dumpStack

static getAllStackTraces

getId

getName // Thread-<number>

getPriority / setPriority

static sleep

Konkret schauen wir uns diese public-Methoden an. Der Übersichtlichkeit halber sind nur die Namen der Methoden auf dieser Folie angegeben sowie die Information, ob die Methode static definiert ist oder nicht, also ob Klassen- oder Objektmethode.

Klasse Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

static currentThread

dumpStack

static getAllStackTraces

getId

getName // Thread-<number>

getPriority / setPriority

static sleep

Die Klassenmethode currentThread hat keine Parameter und liefert ein Thread-Objekt zurück. Dieses Thread-Objekt repräsentiert den Thread, in dem diese Methode aufgerufen wurde.

Klasse Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

static currentThread

dumpStack

static getAllStackTraces

getId

getName // Thread-<number>

getPriority / setPriority

static sleep

Damit wird der Call-Stack auf System.err geschrieben.

***Erinnerung*: In Kapitel 08, speziell im Abschnitt zu Bytedaten, hatten wir gesehen, dass das uns wohlbekannte Objekt System.out per Konvention für normale Ausgaben des Programms, System.err für Fehlerausgaben verwendet wird.**

***Erinnerung*: In Kapitel 05, Abschnitt zu Exceptions, hatten wir die Methode printStackTrace von Klasse Exception gesehen, die im Prinzip dasselbe macht. Der Unterschied ist, dass printStackTrace von Exception den Call-Stack zu einem früheren Zeitpunkt liefert, nämlich als die Exception geworfen wurde, was ja auch offensichtlich sinnvoll ist. Methode dumpStack hingegen gibt den Call-Stack aus, wie er zum Zeitpunkt des Aufrufs von dumpStack war.**

Klasse Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

static currentThread

dumpStack

static getAllStackTraces

getId

getName // Thread-<number>

getPriority / setPriority

static sleep

Diese Methode hingegen liefert die Call-Stacks von *allen* momentan aktiven Threads in dieser Java-Programmausführung. Rückgabe ist eine Map, deren Schlüsselwerte von Klasse Thread sind. Die Information zu einem Thread ist ein Array von StackTraceElement. Ein Objekt von Klasse StackTraceElement repräsentiert ein einzelnes Element im Call-Stack.

***Erinnerung*: In Kapitel 07 gab es einen Abschnitt zu Maps.**

Klasse Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

static currentThread

dumpStack

static getAllStackTraces

getId

getName // Thread-<number>

getPriority / setPriority

static sleep

Jeder Thread hat eine ID vom primitiven Datentyp long, die vom Laufzeitsystem vergeben wird. Solange der Thread aktiv ist, bleibt seine ID gleich und wird auch in derselben Programmausführung nicht noch einmal vergeben. Ist ein Thread beendet, kann es sein, dass die ID für einen anderen Thread nochmals vergeben wird.

Klasse Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

static currentThread

dumpStack

static getAllStackTraces

getId

getName // Thread-<number>

getPriority / setPriority

static sleep

Jeder Thread hat auch einen Namen, der aber nicht einmalig sein muss. Einige Konstruktoren von Klasse Thread haben einen String-Parameter, mit dem man den Namen des neuen Threads spezifizieren kann.

Klasse Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

static currentThread

dumpStack

static getAllStackTraces

getId

getName // Thread-<number>

getPriority / setPriority

static sleep

Verwendet man einen Konstruktor von Klasse Thread, mit dem man *keinen* Namen angeben kann, so wie bei unserem einführenden Beispiel weiter vorne, dann wird der Name so wie hier gezeigt gebildet, wobei für <number> eine ganze Zahl eingesetzt wird.

Klasse Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

static currentThread

dumpStack

static getAllStackTraces

getId

getName // Thread-<number>

getPriority / setPriority

static sleep

Jeder Thread hat einen Prioritätswert vom Typ int. Wird ein Java-Programm gestartet, dann hat der der erste Thread einen Prioritätswert, der vom System und von der Vorrangstellung des Nutzers im System abhängt. Bei Einrichtung jedes neuen Threads wird dieser Prioritätswert übernommen. Dieser Prioritätswert findet sich in der Klassenkonstante NORM_PRIORITY von Klasse Thread.

Der Prioritätswert eines Threads kann später durch Methode setPriority geändert werden, allerdings nur im Rahmen zweier weiterer Klassenkonstanten mit selbsterklärenden Namen, MIN_PRIORITY und MAX_PRIORITY. Bei einem Wert außerhalb dieses Intervalls wird eine IllegalArgumentException geworfen.

Falls der Thread, *in dem* setPriority aufgerufen wird, nicht das Recht hat, den Prioritätswert des Threads, *mit dem* setPriority aufgerufen wird, zu ändern, wird eine SecurityException geworfen.

Da beide Exception-Klassen, IllegalArgumentException und SecurityException, von RuntimeException abgeleitet sind, brauchen sie nicht gefangen zu werden.

Klasse Thread

TECHNISCHE UNIVERSITÄT DARMSTADT

static currentThread

dumpStack

static getAllStackTraces

getId

getName // Thread-<number>

getPriority / setPriority

static sleep

Erinnerung: Bei FopBot hatten wir schon die Klassenmethode sleep verwendet, um den Thread nach jedem Schritt für eine kurze Pause anzuhalten, damit das menschliche Auge den einzelnen Schritten folgen konnte. Der Parameter ist vom Typ long und gibt die Länge der Pause in Millisekunden an.

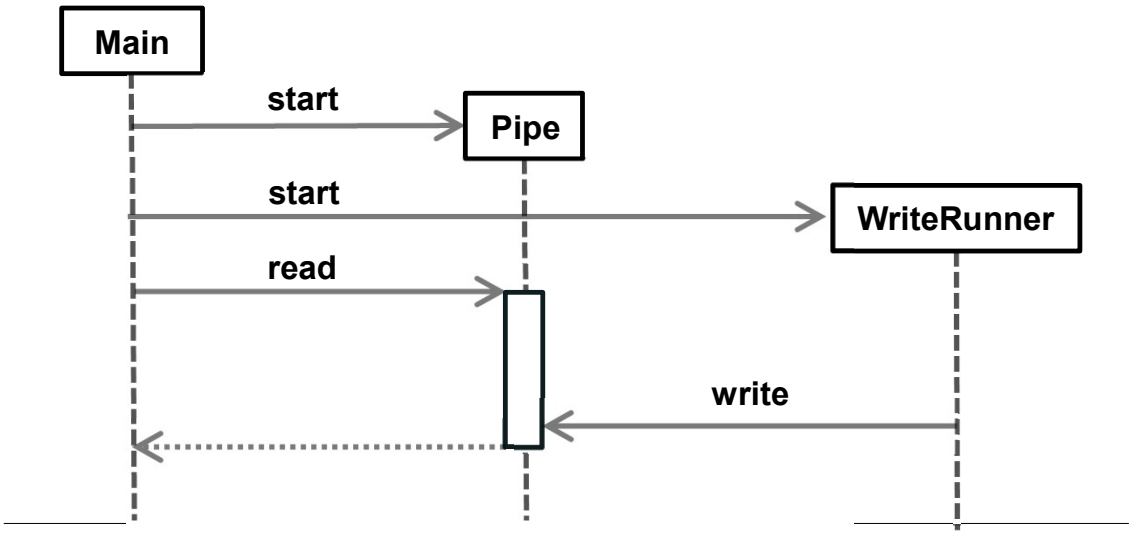
TECHNISCHE UNIVERSITÄT DARMSTADT

Threads und Streams

***Erinnerung*: In Kapitel 08, Abschnitt zu Bytedaten, hatten wir kurz die Klassen PipedInputStream und PipedOutputStream erwähnt, aber nicht besprochen. Das holen wir jetzt nach in dem Kontext, für den die beiden Klassen primär definiert sind, eben Threads.**

Threads und Streams

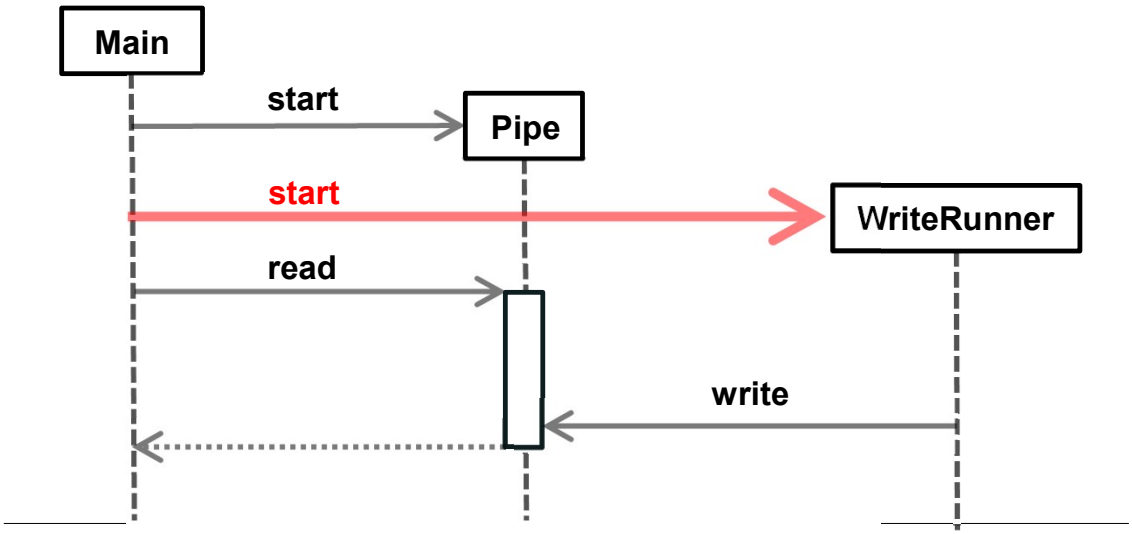
TECHNISCHE UNIVERSITÄT DARMSTADT



Jetzt wird das Gesamtbild im UML-Sequenzdiagramm etwas komplizierter.

Threads und Streams

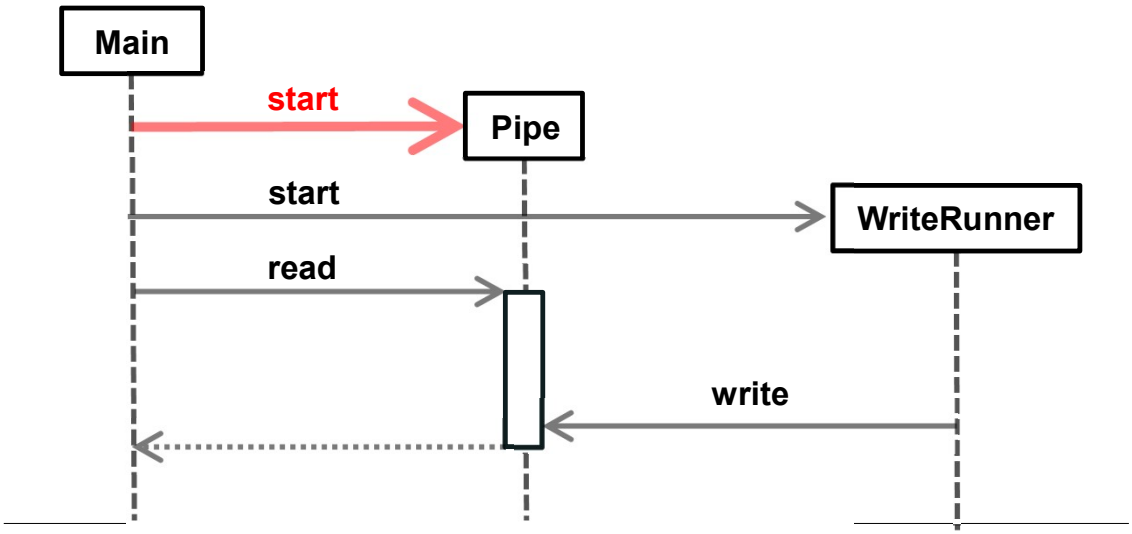
TECHNISCHE UNIVERSITÄT DARMSTADT



Diesen Teil kennen wir schon aus dem ersten Beispiel: Aus einem Thread wird ein anderer gestartet. Letzterer heißt jetzt nicht RandomWriteRunner, sondern WriteRunner, und macht auch etwas anderes als RandomWriteRunner. Klasse WriteRunner haben wir noch zu implementieren, und auch Main muss anders aussehen als im ersten Beispiel.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

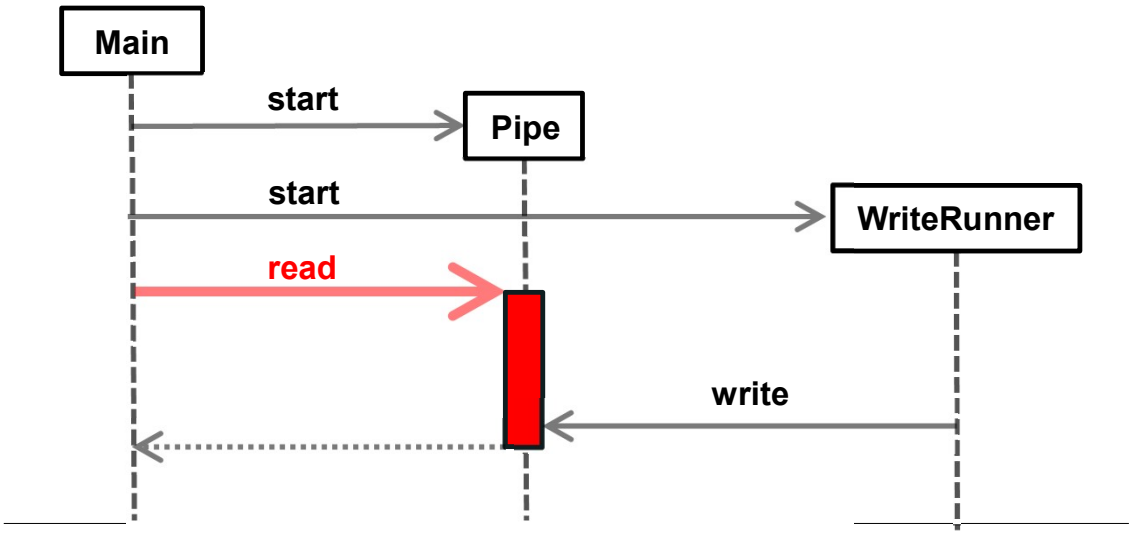


Das ist jetzt neu: Aus Main heraus wird ein dritter Thread gestartet. Generell ist „Pipe“ in der Informatik der Begriff für eine unidirektionale Datenbrücke zwischen zwei Prozessen beziehungsweise Threads. Dabei bedeutet „unidirektional“, dass einer der beiden beteiligten Threads Daten an die Pipe sendet und der andere Thread diese Daten von der Pipe empfängt, aber immer nur in dieser Richtung, nie in die umgekehrte Richtung. Eine Pipe ist also sozusagen eine Einbahnstraße für Daten.

Diese Pipe und diesen Startvorgang müssen wir allerdings nicht selbst implementieren, die sind schon in PipedInputStream und PipedOutputStream implementiert. Wir werden gleich sehen, wie der Umgang damit in Java-Code aussieht.

Threads und Streams

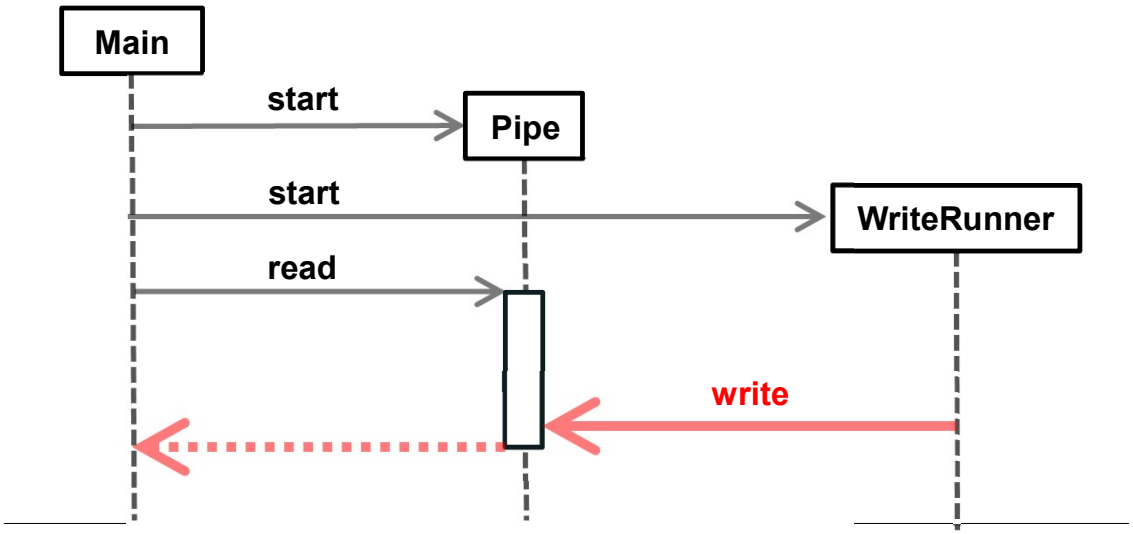
TECHNISCHE UNIVERSITÄT DARMSTADT



Über PipedOutputstream wird eine Nachricht namens read an die Pipe geschickt. Wie wir gleich sehen werden, ist das einfach realisiert durch den Aufruf einer Methode namens read von PipedInputStream. Das rote Rechteck stellt einen aktivierten Zustand des Pipe-Threads dar.

Threads und Streams

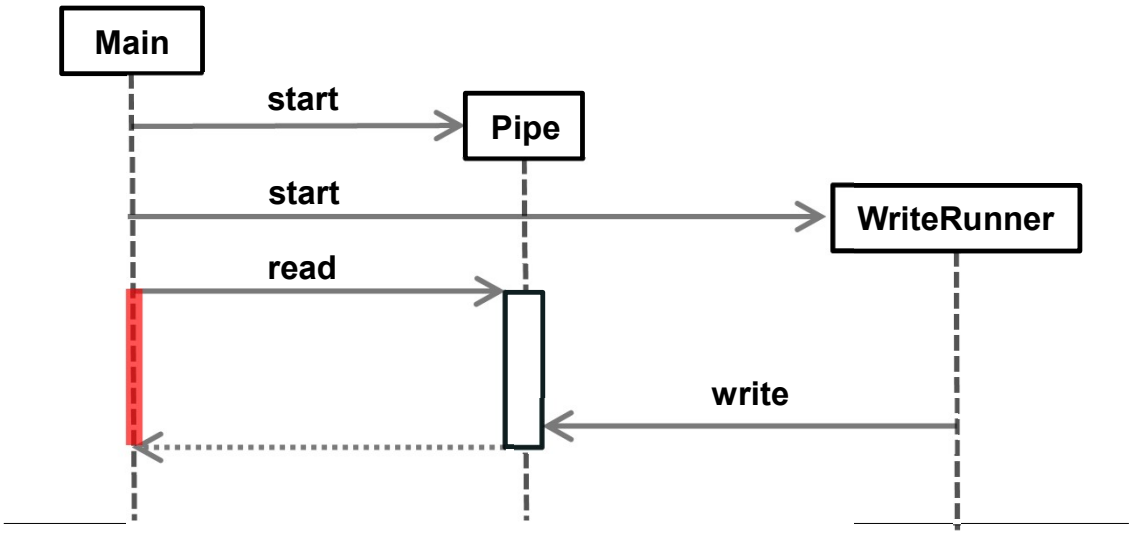
TECHNISCHE UNIVERSITÄT DARMSTADT



Irgendwann sendet WriteRunner Daten mittels einer Nachricht namens write an die Pipe, die analog als Aufruf einer Methode write von PipedOutputStream realisiert ist. Was der WriteRunner in die Pipe geschrieben hat, wird nun von der Pipe als Rückgabewert von read zurückgeliefert. Antworten auf Nachrichten werden in UML-Sequenzdiagrammen gestrichelt dargestellt. Dieser Pfeil stellt also die Antwort auf die Nachricht read dar. Die Antwort erfolgt erst, wenn es etwas zurückzugeben gibt, also nach dem write.

Threads und Streams

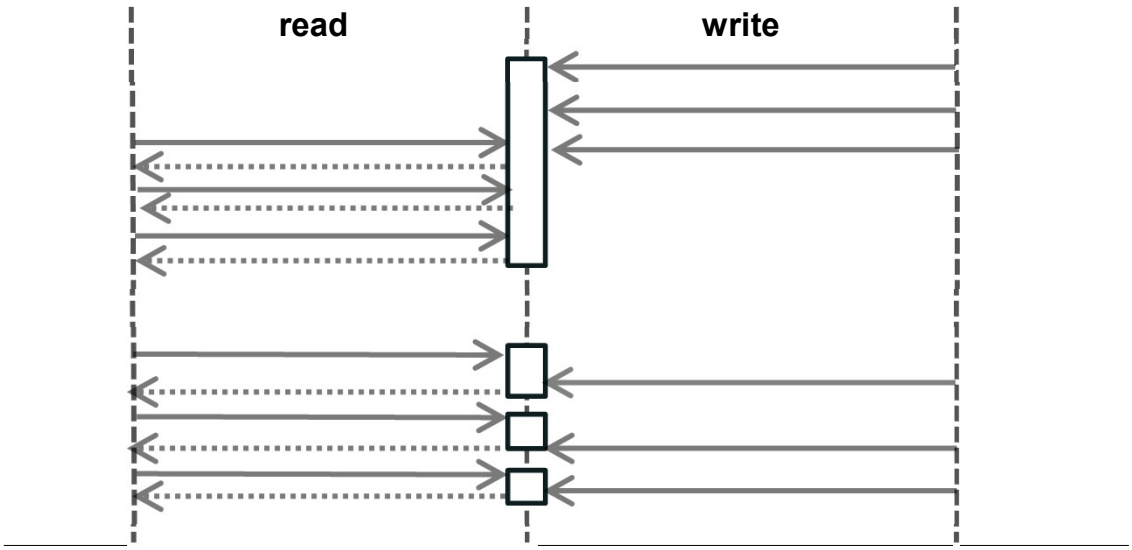
TECHNISCHE UNIVERSITÄT DARMSTADT



In dem Zeitraum, der jetzt rot dargestellt ist, wartet der linke Thread also darauf, dass der Aufruf von read beendet wird, und macht wie üblich in dieser Zeit auch nichts anderes.

Threads und Streams

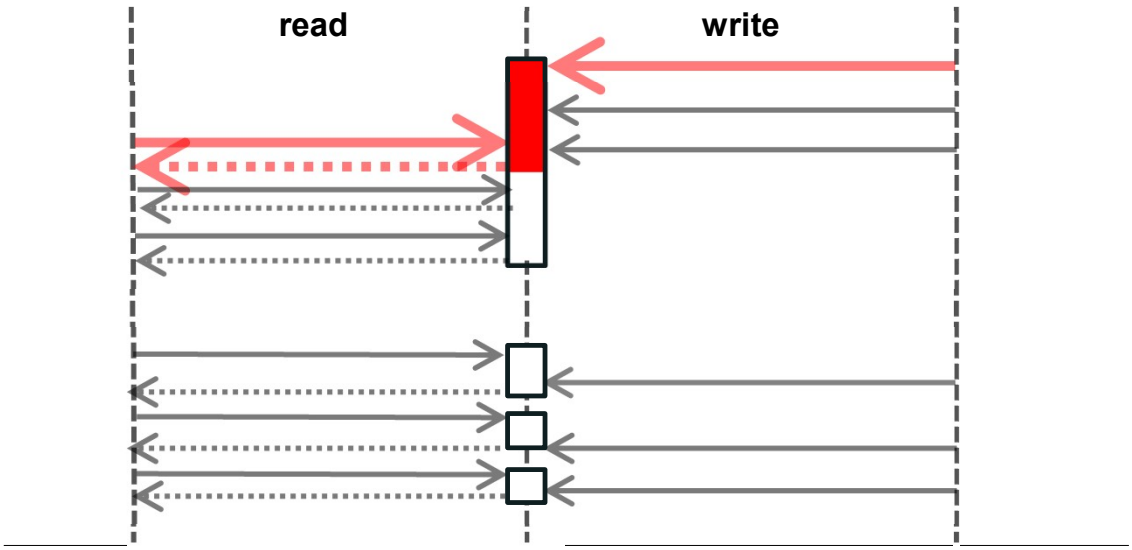
TECHNISCHE UNIVERSITÄT DARMSTADT



Die Anzahl und die Reihenfolge der einzelnen read- und write-Aktionen kann aber beliebig anders sein, tatsächlich kann die Reihenfolge beliebig „chaotisch“ sein. Um das zu illustrieren, sehen wir uns auf dieser Folie eine künstlich konstruierte, aber potentiell durchaus repräsentative kurze Abfolge von read- und write-Aktionen an.

Threads und Streams

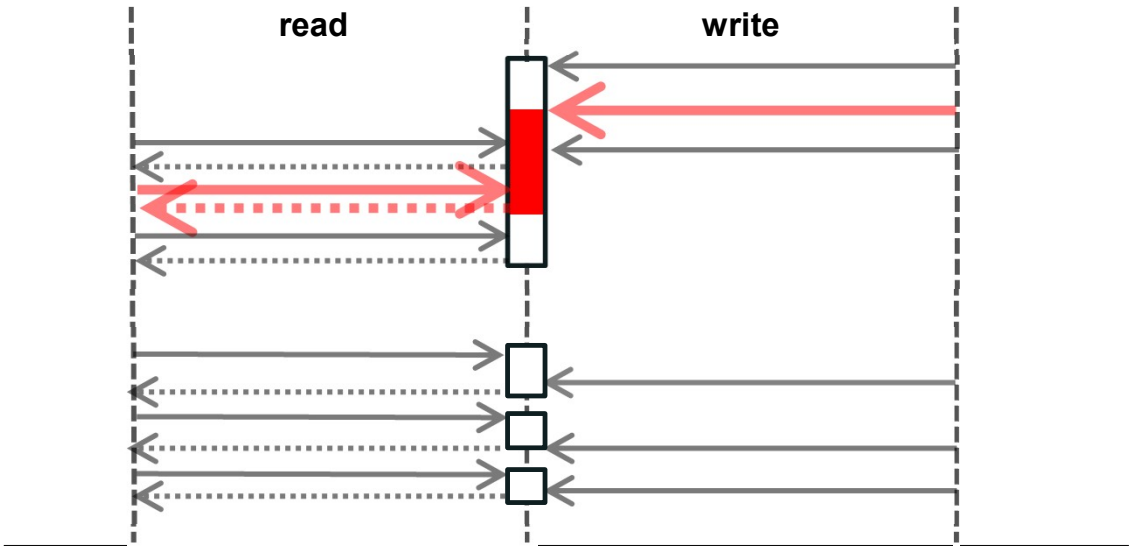
TECHNISCHE UNIVERSITÄT DARMSTADT



Beispielsweise kann es durchaus sein, dass zuerst ein write passiert und erst später ein read. Dann werden die mit write auf die Pipe geschriebenen Daten gepuffert, bis das erste read passiert. Die Antwort auf das read kommt dann natürlich unverzüglich, da nicht mehr auf das write zu warten ist.

Threads und Streams

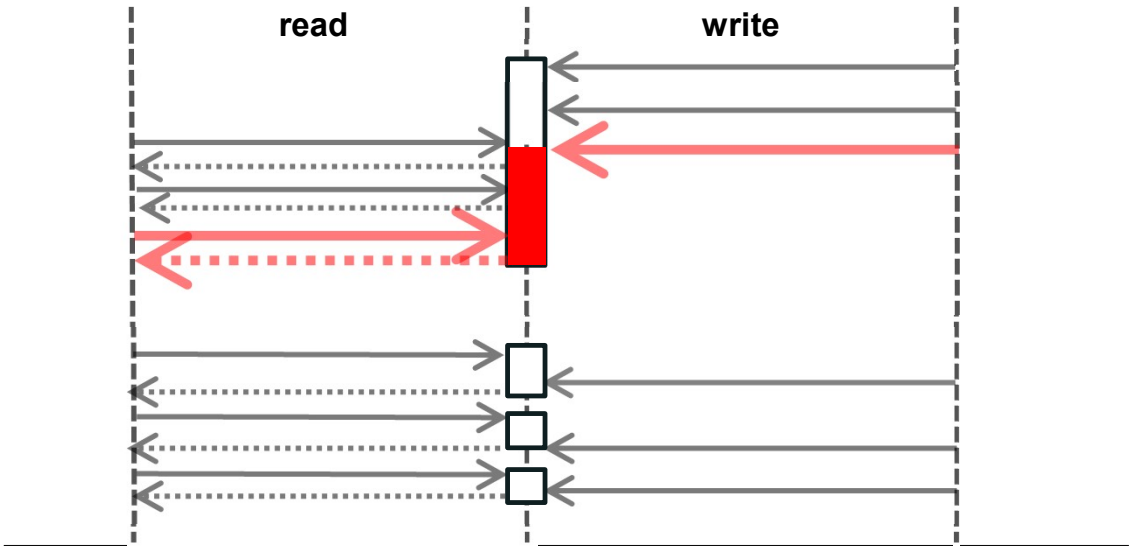
TECHNISCHE UNIVERSITÄT DARMSTADT



Natürlich können auch mehrere write-Aktionen passieren, bevor das erste read passiert. Dann werden die Daten durch die read-Aktionen in derselben Reihenfolge zurückgeliefert, in der sie durch die write-Aktionen in die Pipe geschrieben wurden. Die Daten aus dem zweiten write werden also beim zweiten read zurückgeliefert.

Threads und Streams

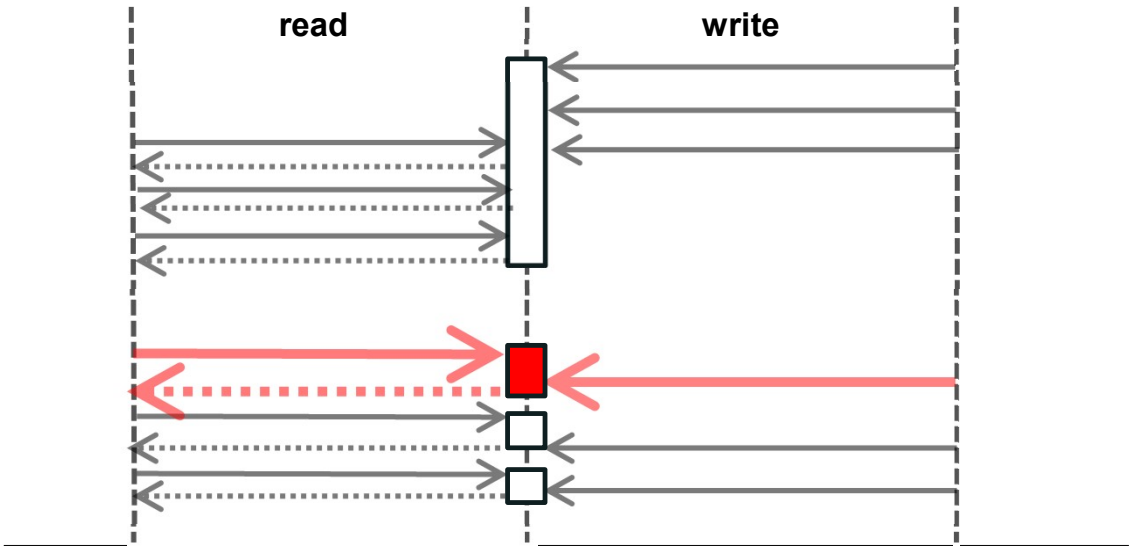
TECHNISCHE UNIVERSITÄT DARMSTADT



Noch ein write-read-Paar. Danach ist der Puffer erst einmal leer, und die Pipe ist in diesem künstlichen Beispiel für kurze Zeit nicht mehr aktiviert im Sinne von UML-Sequenzdiagrammen.

Threads und Streams

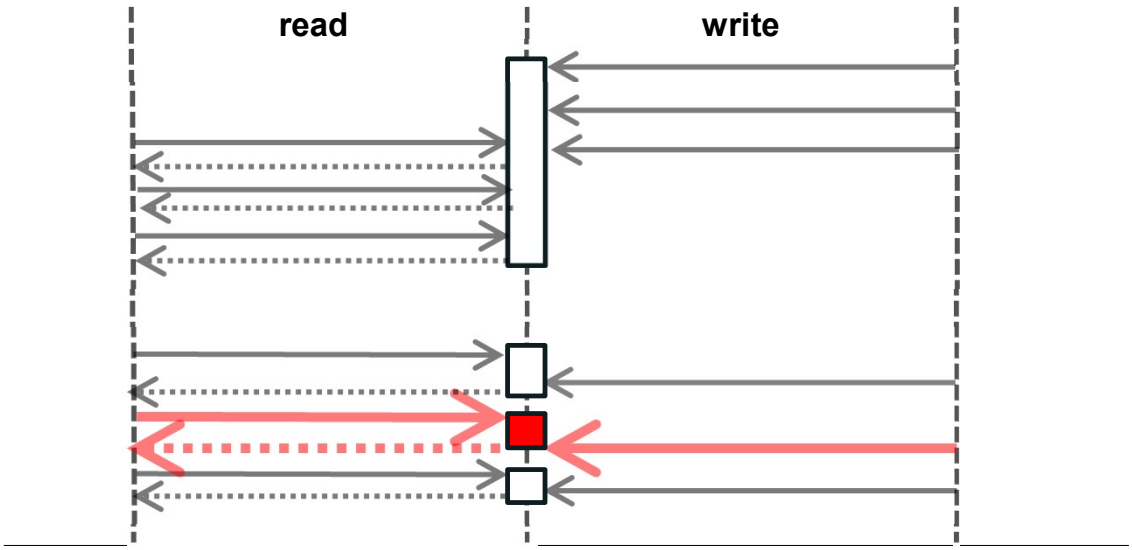
TECHNISCHE UNIVERSITÄT DARMSTADT



Es kann natürlich auch umgekehrt sein: erst ein read, dann ein write. Unverzüglich nach dem write kommt dann die Antwort auf das read.

Threads und Streams

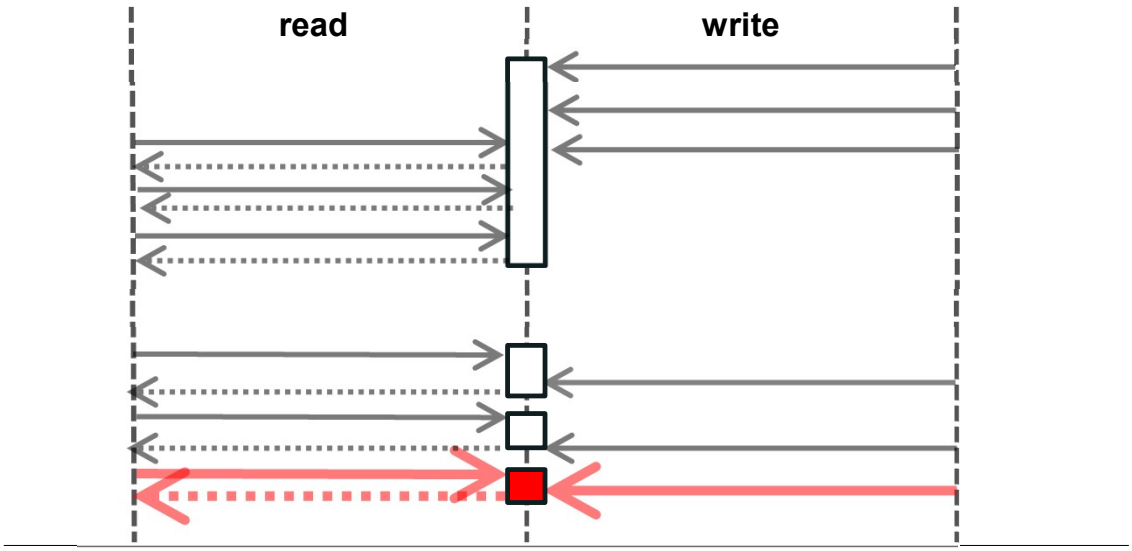
TECHNISCHE UNIVERSITÄT DARMSTADT



Mehrere read-Aktionen vor der ersten write-Aktion können nicht passieren, denn solange keine Antwort auf eine read-Aktion kommt, wartet der Thread, der die read-Aktion gestartet hat. Erst wenn eine Antwort gekommen ist, kann er die nächste read-Aktion starten. Solange jede write-Aktion erst nach der damit korrespondierenden read-Aktion kommt, ist die Reihenfolge also strikt: read – write – Antwort – read – write – Antwort ...

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT



... und so weiter.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
   public void main ( String[ ] args ) {
      PipedOutputStream out = new PipedOutputStream();
      try ( PipedInputStream in = new PipedInputStream ( out ) ) {
         Runnable runnable = new WriteRunner ( out );
         new Thread(runnable).start();
         .........
       } catch ( IOException exc ) { ......... }
   }
}
```

Nun das Ganze in Java-Code.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
   public void main ( String[ ] args ) {
      PipedOutputStream out = new PipedOutputStream();
      try ( PipedInputStream in = new PipedInputStream ( out ) ) {
         Runnable runnable = new WriteRunner ( out );
         new Thread(runnable).start();
         .........
      } catch ( IOException exc ) { ......... }
   }
}
```

Wir richten erst einmal ein PipedOutputStream-Objekt noch ohne Datensenke ein. Dabei wird keine Exception geworfen, die Anweisung kann daher vor dem try-with-ressources stehen.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
   public void main ( String[ ] args ) {
      PipedOutputStream out = new PipedOutputStream();
      try ( PipedInputStream in = new PipedInputStream ( out ) ) {
        Runnable runnable = new WriteRunner ( out );
        new Thread(runnable).start();
        .........
      } catch ( IOException exc ) { ......... }
   }
}
```

Ein PipedInputStream-Objekt lässt sich so mit einem PipedOutputstream-Objekt verknüpfen. Alles, was auf den PipedOutputstream geschrieben wird, kann aus dem PipedInputStream gelesen werden. In der Informatik ist *Pipe* der Fachbegriff für eine solche Verbindung zwischen zwei Prozessen oder Threads, in der der eine schreibt und der andere liest.

Dieser Konstruktor kann eine IOException werfen, daher im try-with-ressources.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
   public void main ( String[ ] args ) {
      PipedOutputStream out = new PipedOutputStream();
      try ( PipedInputStream in = new PipedInputStream ( out ) ) {
        Runnable runnable = new WriteRunner ( out );
        new Thread(runnable).start();
        .....
      } catch ( IOException exc ) { ..... }
   }
}
```

Wir müssen gleich noch eine passende Runnable-Klasse definieren, mit einem Konstruktor, der den einen Teil der Verknüpfung bekommt. Wir werden sie WriteRunner nennen.

Grundsätzlich ist beides gleichermaßen möglich: Der neu eingerichtete Thread schreibt, und der ursprüngliche Thread liest, oder umgekehrt. Wir haben uns hier für ersteres entschieden, im zweiten Fall hätten wir in statt out übergeben.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
   public void main ( String[ ] args ) {
      PipedOutputStream out = new PipedOutputStream();
      try ( PipedInputStream in = new PipedInputStream ( out ) ) {
        Runnable runnable = new WriteRunner ( out );
        new Thread(runnable).start();
        .........
      } catch ( IOException exc ) { ......... }
   }
}
```

An dieser Anweisung hat sich gegenüber dem einführenden Beispiel nichts geändert.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
   public void main ( String[ ] args ) {
      PipedOutputStream out = new PipedOutputStream();
      try ( PipedInputStream in = new PipedInputStream ( out ) ) {
        Runnable runnable = new WriteRunner ( out );
        new Thread(runnable).start();
        .....
      } catch ( IOException exc ) { ..... }
   }
}
```

Diesen Teil werden wir als nächstes füllen.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Reader reader = new InputStreamReader ( in );
LineNumberReader lnr = new LineNumberReader ( reader );
String nextLine;
try {
  while ( ( nextLine = lnr.readLine() ) != null )
     doSomethingWith ( lnr );
}  catch ( IOException exc ) { ..... }
```

Hier sehen Sie die fehlenden Anweisungen.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Reader reader = new InputStreamReader ( in );
LineNumberReader lnr = new LineNumberReader ( reader );
String nextLine;
try {
  while ( ( nextLine = lnr.readLine() ) != null )
     doSomethingWith ( lnr );
}  catch ( IOException exc ) { ..... }
```

Das kennen wir schon aus dem Kapitel zu Streams und Files: aus einem InputStream über InputStreamReader einen LineNumberReader auf derselben Datenquelle einrichten. Da die beiden Konstruktoren keine Exceptions werfen, ist try-with-ressources hier nicht notwendig.

Aus in wird ja das gelesen, was der neue Thread auf Basis von Klasse WriteRunner in seinen PipedInputStream hineinschreibt. Reader und LineNumberReader dienen wieder nur dem einfacheren Zugriff auf die Textdaten, siehe Kapitel 08.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Reader reader = new InputStreamReader ( in );
LineNumberReader lnr = new LineNumberReader ( reader );
String nextLine;
try {
  while ( ( nextLine = lnr.readLine() ) != null )
     doSomethingWith ( lnr );
}  catch ( IOException exc ) { ..... }
```

Alles, was im neu eingerichteten Thread auf den OutputStream geschrieben wird, wird also hier nun zeilenweise in den String nextLine ausgelesen.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Reader reader = new InputStreamReader ( in );
LineNumberReader lnr = new LineNumberReader ( reader );
String nextLine;
try {
  while ( ( nextLine = lnr.readLine() ) != null )
     doSomethingWith ( lnr );
}  catch ( IOException exc ) { ..... }
```

***Erinnerung*: In Kapitel 01b, Abschnitt zur Bindungsstärke von Operatoren, hatten wir gesehen, dass die zuweisungsbasierten Operatoren schwächer als die Vergleichsoperatoren binden. Daher muss hier die Zuweisung in Klammern gesetzt werden, denn wir wollen ja gerade umgekehrt, dass *zuerst* die Zuweisung ausgeführt wird und *danach* erst der Test auf Ungleichheit.**

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Reader reader = new InputStreamReader ( in );
LineNumberReader lnr = new LineNumberReader ( reader );
String nextLine;
try {
  while ( ( nextLine = lnr.readLine() ) != null )
     doSomethingWith ( lnr );
}  catch ( IOException exc ) { ..... }
```

***Erinnerung*: In Kapitel 03c, Abschnitt „Ausdrücke haben Typ und Wert und optional Seiteneffekte“, hatten wir zudem gesehen, dass diese Zuweisung nicht nur den Seiteneffekt hat, dass nextLine die nächste Zeile aus dem InputStream zugewiesen wird; darüber hinaus hat die Zuweisung auch einen Rückgabewert, nämlich den zugewiesenen Wert. Dieser wird im Kopf dieser while-Schleife also auf ungleich null getestet, was auch genau richtig ist, denn Rückgabe null bedeutet, dass das Ende der Datenquelle erreicht ist.**

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class WriteRunner implements Runnable {
   PipedOutputStream pout;
   public WriteRunner ( PipedOutputStream pout ) {
      this.pout = pout;
   }
   public void run() {
      for ( int i = 0; i < 1000; i++ )
         pout.println ( i );
   }
}
```

Es fehlt noch die Definition der Runnable-Klasse WriteRunner, die die Daten in den PipedInputStream hineinschreibt.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class WriteRunner implements Runnable {
   PipedOutputStream pout;
   public WriteRunner ( PipedOutputStream pout ) {
      this.pout = pout;
   }
   public void run() {
      for ( int i = 0; i < 1000; i++ )
         pout.println ( i );
   }
}
```

Der OutputStream wird im Konstruktor als Attribut gespeichert und in Methode run verwendet.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

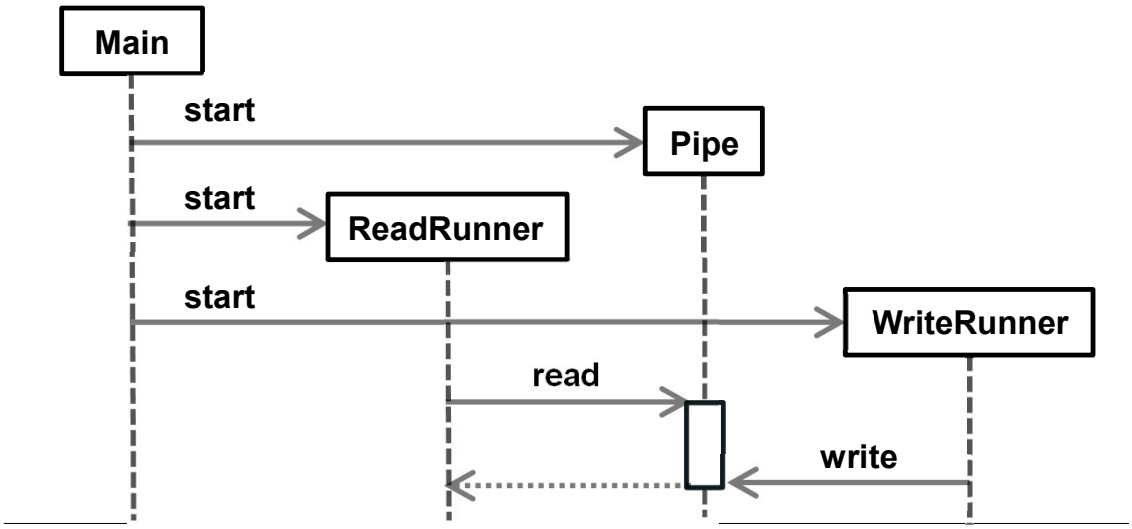
```
public class WriteRunner implements Runnable {
   PipedOutputStream pout;
   public WriteRunner ( PipedOutputStream pout ) {
      this.pout = pout;
   }
   public void run() {
      for ( int i = 0; i < 1000; i++ )
         pout.println ( i );
   }
}
```

In Methode run werden beispielhaft eintausend Zeilen auf den OutputStream geschrieben. Im ursprünglichen Thread werden also genau diese eintausend Zeilen aus dem InputStream gelesen.

***Erinnerung*: Werte wie die eintausend hier sollte man bekanntlich nicht im Programm explizit hinschreiben, sondern dafür eine Konstante mit einem sprechenden Namen definieren. Aber hier geht es ja nur um Illustration.**

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT



Jetzt gehen wir noch einen Schritt weiter und erzeugen *zwei* neue Threads, die miteinander kommunizieren und ansonsten unabhängig voneinander und vom ursprünglichen Thread laufen. Unten sehen Sie nur ein einzelnes Beispiel für read-Aktion, write-Aktion und Antwort.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public void main ( String[ ] args ) {
    PipedInputStream pin = .....;
    try ( PipedOutputStream pout = new PipedOutputStream ( pin ) ) {
      Runnable runnable1 = new ReadRunner ( pin );
      Runnable runnable2 = new WriteRunner ( pout );
      new Thread(runnable1).start();
      new Thread(runnable2).start();
    } catch ( IOException exc ) { ..... }
  }
}
```

Nun der Java-Code dazu.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public void main ( String[ ] args ) {
    PipedInputStream pin = .....;
    try ( PipedOutputStream pout = new PipedOutputStream ( pin ) ) {
      Runnable runnable1 = new ReadRunner ( pin );
      Runnable runnable2 = new WriteRunner ( pout );
      new Thread(runnable1).start();
      new Thread(runnable2).start();
    } catch ( IOException exc ) { ..... }
  }
}
```

Wie bisher erzeugen wir einen PipedInputStream und einen PipedOutputStream, wobei die Verknüpfung beider auch hier wieder in ein try-with-ressources muss, da potentiell eine IOException geworfen wird und dann ja auch gefangen werden muss, wenn sie wie hier nicht weitergereicht wird.

Im vorangegangenen Beispiel hatten wir zuerst den PipedOutputStream und danach den PipedInputStream nebst Verknüpfung eingerichtet; hier sehen Sie jetzt, dass das umgekehrt genauso geht. Das Ergebnis ist dasselbe.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public void main ( String[ ] args ) {
    PipedInputStream pin = .....;
    try ( PipedOutputStream pout = new PipedOutputStream ( pin ) ) {
      Runnable runnable1 = new ReadRunner ( pin );
      Runnable runnable2 = new WriteRunner ( pout );
      new Thread(runnable1).start();
      new Thread(runnable2).start();
    } catch ( IOException exc ) { ..... }
  }
}
```

Für die beiden Threads brauchen wir je ein Runnable-Objekt.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public void main ( String[ ] args ) {
    PipedInputStream pin = .....;
    try ( PipedOutputStream pout = new PipedOutputStream ( pin ) ) {
      Runnable runnable1 = new ReadRunner ( pin );
      Runnable runnable2 = new WriteRunner ( pout );
      new Thread(runnable1).start();
      new Thread(runnable2).start();
    } catch ( IOException exc ) { ..... }
  }
}
```

Die Klasse ReadRunner macht das, was im vorangehenden Beispiel der ursprüngliche Thread gemacht hat: zeilenweise vom PipedInputStream lesen. Wir werden diese Klasse hier nicht implementieren; ihre Implementation sollte nach dem bisher Gesagten eine einfache Aufgabe sein.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public void main ( String[ ] args ) {
    PipedInputStream pin = .....;
    try ( PipedOutputStream pout = new PipedOutputStream ( pin ) ) {
      Runnable runnable1 = new ReadRunner ( pin );
      Runnable runnable2 = new WriteRunner ( pout );
      new Thread(runnable1).start();
      new Thread(runnable2).start();
    } catch ( IOException exc ) { ..... }
  }
}
```

Wie gehabt, richten wir beide Threads ein, starten sie und vergessen sie. Die beiden Threads kommunizieren miteinander über die oben eingerichtete Verknüpfung der beiden Streams, völlig entkoppelt vom ursprünglichen Thread.

TECHNISCHE UNIVERSITÄT DARMSTADT

Interferierende Threads

Wenn mehrere Threads auf dieselbe Ressource zugreifen, kann man nicht vorhersagen, in welcher Reihenfolge die Zugriffe sein werden. Die Reihenfolge kann auch bei zwei aufeinanderfolgenden Programmläufen desselben Programms unterschiedlich sein.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public void main ( String[ ] args ) {
     Runnable runnable = new WriteStdOutRunner();
     new Thread(runnable).start();
     for ( int i = 0; i < 1000; i++ )
         System.out.println ( i );
  }
}
```

Wir sehen uns das an einer kleinen Variation unserer bisherigen Beispiele an.

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public void main ( String[ ] args ) {
     Runnable runnable = new WriteStdOutRunner();
     new Thread(runnable).start();
     for ( int i = 0; i < 1000; i++ )
         System.out.println ( i );
  }
}
```

Klasse WriteRunner ist durch eine leicht abgewandelte Runnable-Klasse ersetzt. Diese schreibt jetzt nicht mehr auf einen PipedOutputStream, sondern auf System.out. Der Konstruktor braucht dann – im Gegensatz zu WriteRunner – auch keinen Parameter, um den Stream zu spezifizieren.

***Nebenbemerkung*: System.out wird Standard-Output genannt, daher der farblich unterlegte Name, analog Standard-Input für System.in und Standard-Error für System.err. Diese Begriffe werden auch bei den analogen Konstrukten in anderen Programmiersprachen verwendet, zum Beispiel stdout, stdin und stderr in C und cout, cin und cerr in C++.**

Threads und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Main {
  public void main ( String[ ] args ) {
    Runnable runnable = new WriteStdOutRunner();
    new Thread(runnable).start();
    for ( int i = 0; i < 1000; i++ )
        System.out.println ( i );
  }
}
```

Das Problem taucht jetzt hier auf: Der ursprüngliche und der neue Thread schreiben simultan auf denselben Stream. Die Ausgaben der beiden Threads erscheinen potentiell beliebig vermischt auf der Datensenke. Wie diese Vermischung genau aussieht, ist reiner Zufall und kann auch von Lauf zu Lauf des Java-Programms völlig unterschiedlich sein.

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallelisierung mit Threads

Viele Prozesse mit hoher Laufzeit lassen sich gut in Teile zerlegen, die völlig unabhängig voneinander abgearbeitet werden können. Diese Situation findet sich häufig bei numerischen Berechnungen oder auch bei Datenbankoperationen.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double[ ] a = new double [ ..... ];
for ( int i = 0; i < a.length; i++ ) {
   a[i] = 2 + 3 / i;
}
double[ ] b = new double [ a.length ];
for ( int i = 0; i < a.length; i++ )
   b[i] = X.someHighlyTimeConsumingFunction ( a[i] );
```

Dazu wieder ein einfaches, illustratives Beispiel, auf dieser Folie erst einmal noch *ohne* Parallelisierung.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double[ ] a = new double [ ..... ];
for ( int i = 0; i < a.length; i++ ) {
   a[i] = 2 + 3 / i;
}
double[ ] b = new double [ a.length ];
for ( int i = 0; i < a.length; i++ )
   b[i] = X.someHighlyTimeConsumingFunction ( a[i] );
```

Das Array a wird mit irgendwelchen belanglosen Beispieldaten gefüllt.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double[ ] a = new double [ ..... ];
for ( int i = 0; i < a.length; i++ ) {
   a[i] = 2 + 3 / i;
}
double[ ] b = new double [ a.length ];
for ( int i = 0; i < a.length; i++ )
   b[i] = X.someHighlyTimeConsumingFunction ( a[i] );
```

Die beiden Arrays a und b haben dieselbe Länge, das heißt, die Elemente von a und b stehen in Eins-zu-eins-Korrespondenz zueinander.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double[ ] a = new double [ ..... ];
for ( int i = 0; i < a.length; i++ ) {
   a[i] = 2 + 3 / i;
}
double[ ] b = new double [ a.length ];
for ( int i = 0; i < a.length; i++ )
   b[i] = X.someHighlyTimeConsumingFunction ( a[i] );
```

Jedes Element in b wird allein aus dem entsprechenden Element in a berechnet, zum Beispiel eine mathematische Funktion. Die einzelnen Elemente werden also völlig unabhängig voneinander berechnet, und damit ist die bestmögliche Ausgangssituation für Parallelisierung gegeben.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface DoubleToDoubleFunction {
  double apply ( double x );
}
```

X::someHighlyTimeConsumingFunction

Wir gehen aber gleich einen Schritt weiter und machen die Funktion, die jedes Element in b aus dem entsprechenden Element in a berechnet, austauschbar, und zwar genau mit dem Mechanismus, den wir im Kapitel 04c im gleichnamigen Abschnitt eingeführt und seitdem immer wieder dafür verwendet haben: Functional Interfaces und Lambda-Ausdrücke.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface DoubleToDoubleFunction {
  double apply ( double x );
}
```

X::someHighlyTimeConsumingFunction

Erinnerung: In Kapitel 04c, Abschnitt zu Functional Interfaces und Lambda-Ausdrücken, haben wir gesehen, dass Package java.util.function diverse Interfaces ähnlich zu diesem enthält.

Aber genau dieses Interface, das Funktionen repräsentiert, die den primitiven Datentyp double wieder in double abbildet, ist nicht vordefiniert, daher hatten wir es an verschiedenen Stellen schnell selbst definiert.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface DoubleToDoubleFunction {
  double apply ( double x );
}
```

X::someHighlyTimeConsumingFunction

Sinnvollerweise hat die funktionale Methode bei uns dann auch denselben Namen wie bei den entsprechenden Interfaces in Package java.util.functional.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface DoubleToDoubleFunction {
  double apply ( double x );
}
```

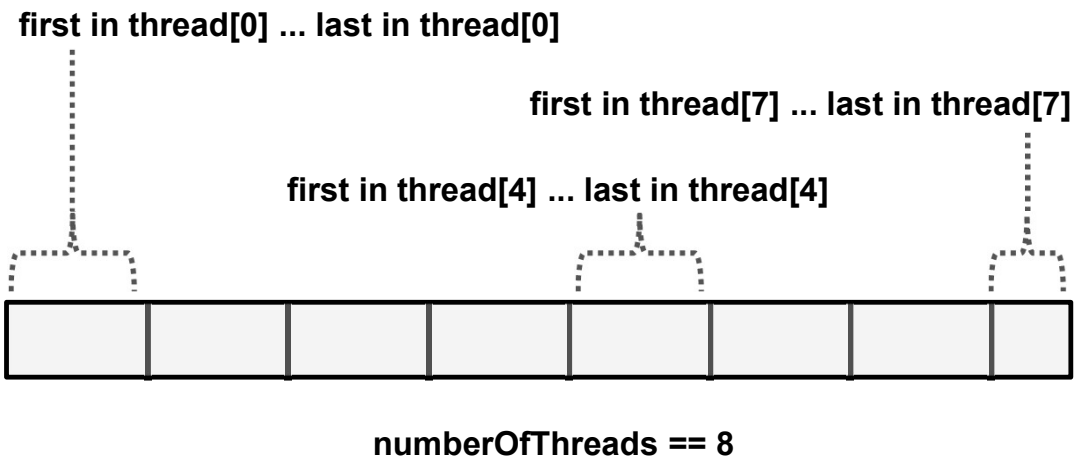
```
X::someHighlyTimeConsumingFunction
```

***Erinnerung*: In Kapitel 08, Abschnitt zu Methodennamen als Lambda-Ausdrücken, hatten wir schon diese verkürzte Schreibweise gesehen für Lambda-Ausdrücke, die nur aus einem einzelnen Methodenaufruf bestehen.**

Diese uns nicht näher bekannte Methode soll ja einen Parameter vom Typ double haben und auch double zurückliefern. Daher passt dieser Lambda-Ausdruck auf das Interface DoubleToDoubleFunction.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT



Nach dieser kleinen Vorarbeit können wir jetzt darangehen, die Berechnung der einzelnen Elemente von b aus dem jeweils korrespondierenden Element von a auf einzelne Threads zu verteilen. Hier sehen Sie das Schema: Jeder einzelne Thread bearbeitet ein Segment des Indexbereichs.

Jedes Segment ist im Prinzip gleich groß. Allerdings ist die gesamte Arraylänge nicht unbedingt ein ganzzahliges Vielfaches der Anzahl Threads. Daher ist in der Regel das letzte Segment kürzer.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class DoubleArraySegmentRunner implements Runnable {
    private double[ ] a;
    private double[ ] b;
    private int first;
    private int last;
    private DoubleToDoubleFunction fct;
    .........
}
```

Nun der Java-Code dazu.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class DoubleArraySegmentRunner implements Runnable {
    private double[ ] a;
    private double[ ] b;
    private int first;
    private int last;
    private DoubleToDoubleFunction fct;
    .........
}
```

Wie immer, bemühen wir uns um einen aussagekräftigen Namen, der möglichst alle Aspekte berücksichtigt. Die Idee ist, dass jeder Thread ein separates Segment des Arrays verarbeitet.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class DoubleArraySegmentRunner implements Runnable {
  private double[ ] a;
  private double[ ] b;
  private int first;
  private int last;
  private DoubleToDoubleFunction fct;
  .........
}
```

Die beiden Arrays, a und b, werden wir gleich im Konstruktor übergeben und in diesen beiden Attributen speichern.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class DoubleArraySegmentRunner implements Runnable {
   private double[ ] a;
   private double[ ] b;
   private int first;
   private int last;
   private DoubleToDoubleFunction fct;
   .........
}
```

Jedem einzelnen Thread wird ein eigenes Segment von Index first bis Index last in den beiden Arrays zugeordnet, und zwar so, dass jeder Index in genau einem dieser Indexbereiche ist. In der Mathematik und in der Informatik sagt man, der Indexbereich der beiden Arrays wird in Unterbereiche oder Segmente *partitioniert*.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class DoubleArraySegmentRunner implements Runnable {
   private double[ ] a;
   private double[ ] b;
   private int first;
   private int last;
   private DoubleToDoubleFunction fct;
   .........
}
```

Und hier wird die auszuführende Funktion gespeichert, die jede Komponente von b aus der Komponente von a am selben Index berechnen soll.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class DoubleArraySegmentRunner implements Runnable {
   private double[ ] a;
   private double[ ] b;
   private int first;
   private int last;
   private DoubleToDoubleFunction fct;
   ……..
}
```

Alles Weitere dann auf den nächsten Folien.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public DoubleArraySegmentRunner
                ( double[ ] a, double[ ] b, int first, int last,
                  DoubleToDoubleFunction fct ) {
    this.a = a;
    this.b = b;
    this.first = first;
    this.last = last;
    this.fct = fct;
}
```

Der Konstruktor hat einfach einen Parameter desselben Namens für jedes Attribut und initialisiert die Attribute mit diesen Parametern.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void run () {
  for ( int i = first; i <= last; i++ )
       b[i]  = fct.apply ( a[i] );
}
```

Die Methode run der neuen Runnable-Klasse DoubleArraySegmentRunner besteht einfach darin, dass die Indizes von first bis last bearbeitet werden.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int numberOfThreads = Runtime.getRunTime().availableProcessors() – 1;
int segmentSize = (int) Math.ceil ( (double)a.length / numberOfThreads );
DoubleToDoubleFunction fct = X::someHighlyTimeConsumingFunction;
Thread[ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) {
   int first = i * segmentSize;
   int last = Math.min ( ( i + 1 ) * segmentSize, a.length ) – 1;
   threads[i] = new Thread(new DoubleArraySegmentRunner(a,b,first,last,fct));
   threads[i].start();
}
.....
```

Jetzt wenden wir uns wieder dem ursprünglichen Thread zu, der in der einführenden Version das gesamte Array selbst bearbeitet hat und nun statt dessen die Aufteilung der Arbeitslast auf Threads realisieren soll.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int numberOfThreads = Runtime.getRunTime().availableProcessors() – 1;
int segmentSize = (int) Math.ceil ( (double)a.length / numberOfThreads );
DoubleToDoubleFunction fct = X::someHighlyTimeConsumingFunction;
Thread[ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) {
   int first = i * segmentSize;
   int last = Math.min ( ( i + 1 ) * segmentSize, a.length ) – 1;
   threads[i] = new Thread(new DoubleArraySegmentRunner(a,b,first,last,fct));
   threads[i].start();
}
.....
```

Die Aufteilung der Arbeitslast macht ja nur Sinn auf einer Hardwareplattform mit mehreren Prozessoren. Die Anzahl der Prozessoren kann man abfragen so wie hier gezeigt. Klasse Runtime in Package java.lang hat eine Klassenmethode getRunTime, die einen Verweis auf ein Objekt von Klasse Runtime zurückliefert. Dieses Objekt repräsentiert die Laufzeitumgebung des Threads, in dem die Methode aufgerufen wird. Die Objektmethode availableProcessors von Runtime liefert die Anzahl der Prozessoren als int-Wert.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int numberOfThreads = Runtime.getRunTime().availableProcessors() – 1;
int segmentSize = (int) Math.ceil ( (double)a.length / numberOfThreads );
DoubleToDoubleFunction fct = X::someHighlyTimeConsumingFunction;
Thread[ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) {
   int first = i * segmentSize;
   int last = Math.min ( ( i + 1 ) * segmentSize, a.length ) – 1;
   threads[i] = new Thread(new DoubleArraySegmentRunner(a,b,first,last,fct));
   threads[i].start();
}
.....
```

Der ursprüngliche Thread braucht natürlich ebenfalls einen Prozessor, also minus 1.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int numberOfThreads = Runtime.getRunTime().availableProcessors() – 1;
int segmentSize = (int) Math.ceil ( (double)a.length / numberOfThreads );
DoubleToDoubleFunction fct = X::someHighlyTimeConsumingFunction;
Thread[ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) {
   int first = i * segmentSize;
   int last = Math.min ( ( i + 1 ) * segmentSize, a.length ) – 1;
   threads[i] = new Thread(new DoubleArraySegmentRunner(a,b,first,last,fct));
   threads[i].start();
}
.....
```

Die Größe eines einzelnen Segments, das einem einzelnen Thread zugewiesen wird, bestimmt sich aus der Größe des Arrays und der Anzahl einzurichtender Threads.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int numberOfThreads = Runtime.getRunTime().availableProcessors() – 1;
int segmentSize = (int) Math.ceil ( (double)a.length / numberOfThreads );
DoubleToDoubleFunction fct = X::someHighlyTimeConsumingFunction;
Thread[ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) {
   int first = i * segmentSize;
   int last = Math.min ( ( i + 1 ) * segmentSize, a.length ) – 1;
   threads[i] = new Thread(new DoubleArraySegmentRunner(a,b,first,last,fct));
   threads[i].start();
}
.....
```

In dem speziellen Fall, dass die Anzahl der Threads ein Teiler der Arraylänge ist, würde ganzzahlige Division die korrekte Segmentgröße liefern.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int numberOfThreads = Runtime.getRunTime().availableProcessors() – 1;
int segmentSize = (int) Math.ceil ( (double)a.length / numberOfThreads );
DoubleToDoubleFunction fct = X::someHighlyTimeConsumingFunction;
Thread[ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) {
   int first = i * segmentSize;
   int last = Math.min ( ( i + 1 ) * segmentSize, a.length ) – 1;
   threads[i] = new Thread(new DoubleArraySegmentRunner(a,b,first,last,fct));
   threads[i].start();
}
.....
```

Aber im Allgemeinen bleibt ja ein Rest bei ganzzahliger Division. Deshalb muss nichtganzzahlige Division angewendet und das Ergebnis aufgerundet werden.

Nichtganzzahlige Division wird angewendet, wenn mindestens einer der beiden Operanden nichtganzzahlig ist. Das erzwingt man mit expliziter Konversion eines der beiden Operanden, in diesem Fall des Dividenden.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int numberOfThreads = Runtime.getRunTime().availableProcessors() – 1;
int segmentSize = (int) Math.ceil ( (double)a.length / numberOfThreads );
DoubleToDoubleFunction fct = X::someHighlyTimeConsumingFunction;
Thread[ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) {
   int first = i * segmentSize;
   int last = Math.min ( ( i + 1 ) * segmentSize, a.length ) – 1;
   threads[i] = new Thread(new DoubleArraySegmentRunner(a,b,first,last,fct));
   threads[i].start();
}
.....
```

Die Klasse Math hat eine Klassenmethode ceil, die einen Parameter vom Typ double bekommt und auf die nächstgrößere ganze Zahl aufrundet.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int numberOfThreads = Runtime.getRunTime().availableProcessors() – 1;
int segmentSize = (int) Math.ceil ( (double)a.length / numberOfThreads );
DoubleToDoubleFunction fct = X::someHighlyTimeConsumingFunction;
Thread[ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) {
   int first = i * segmentSize;
   int last = Math.min ( ( i + 1 ) * segmentSize, a.length ) – 1;
   threads[i] = new Thread(new DoubleArraySegmentRunner(a,b,first,last,fct));
   threads[i].start();
}
.....
```

Allerdings liefert die Methode ceil von Math double zurück, obwohl das Ergebnis eine ganze Zahl ist.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int numberOfThreads = Runtime.getRunTime().availableProcessors() – 1;
int segmentSize = (int) Math.ceil ( (double)a.length / numberOfThreads );
DoubleToDoubleFunction fct = X::someHighlyTimeConsumingFunction;
Thread[ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) {
   int first = i * segmentSize;
   int last = Math.min ( ( i + 1 ) * segmentSize, a.length ) – 1;
   threads[i] = new Thread(new DoubleArraySegmentRunner(a,b,first,last,fct));
   threads[i].start();
}
.....
```

Wie ein Functional Interface durch einen Lambda-Ausdruck initialisiert werden kann, haben wir schon öfters gesehen, und auch die Kurzschreibweise für Lambda-Ausdrücke, die nur aus einem einzelnen Methodenaufruf bestehen.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int numberOfThreads = Runtime.getRunTime().availableProcessors() – 1;
int segmentSize = (int) Math.ceil ( (double)a.length / numberOfThreads );
DoubleToDoubleFunction fct = X::someHighlyTimeConsumingFunction;
Thread[ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) {
   int first = i * segmentSize;
   int last = Math.min ( ( i + 1 ) * segmentSize, a.length ) – 1;
   threads[i] = new Thread(new DoubleArraySegmentRunner(a,b,first,last,fct));
   threads[i].start();
}
.....
```

Im Gegensatz zu den bisherigen Beispielen wollen wir diesmal noch etwas von den Threads, nachdem wir sie gestartet haben, daher müssen wir für spätere Verwendung Verweise auf die Thread-Objekte speichern. Da die Anzahl der Threads plattformabhängig und somit unvorhersehbar ist, verwenden wir dafür ein Array von Threads.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int numberOfThreads = Runtime.getRunTime().availableProcessors() – 1;
int segmentSize = (int) Math.ceil ( (double)a.length / numberOfThreads );
DoubleToDoubleFunction fct = X::someHighlyTimeConsumingFunction;
Thread[ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) {
   int first = i * segmentSize;
   int last = Math.min ( ( i + 1 ) * segmentSize, a.length ) – 1;
   threads[i] = new Thread(new DoubleArraySegmentRunner(a,b,first,last,fct));
   threads[i].start();
}
.....
```

Nach allen diesen Vorbereitungen geht es jetzt an das Einrichten der einzelnen Threads.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int numberOfThreads = Runtime.getRunTime().availableProcessors() – 1;
int segmentSize = (int) Math.ceil ( (double)a.length / numberOfThreads );
DoubleToDoubleFunction fct = X::someHighlyTimeConsumingFunction;
Thread[ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) {
   int first = i * segmentSize;
   int last = Math.min ( ( i + 1 ) * segmentSize, a.length ) – 1;
   threads[i] = new Thread(new DoubleArraySegmentRunner(a,b,first,last,fct));
   threads[i].start();
}
.....
```

Hier werden auf Basis der Segmentgröße der erste und der letzte Index des jeweiligen Segments berechnet.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int numberOfThreads = Runtime.getRunTime().availableProcessors() – 1;
int segmentSize = (int) Math.ceil ( (double)a.length / numberOfThreads );
DoubleToDoubleFunction fct = X::someHighlyTimeConsumingFunction;
Thread[ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) {
   int first = i * segmentSize;
   int last = Math.min ( ( i + 1 ) * segmentSize, a.length ) – 1;
   threads[i] = new Thread(new DoubleArraySegmentRunner(a,b,first,last,fct));
   threads[i].start();
}
.....
```

Beim letzten Index ist zu beachten, dass das allerletzte Segment eine Größe haben kann, die kleiner als segmentSize ist. Hier muss dafür gesorgt werden, dass der Gesamtindexbereich des Arrays nicht überschritten wird, daher Minimumsbildung mit dem letzten validen Arrayindex.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int numberOfThreads = Runtime.getRunTime().availableProcessors() – 1;
int segmentSize = (int) Math.ceil ( (double)a.length / numberOfThreads );
DoubleToDoubleFunction fct = X::someHighlyTimeConsumingFunction;
Thread[ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) {
   int first = i * segmentSize;
   int last = Math.min ( ( i + 1 ) * segmentSize, a.length ) – 1;
   threads[i] = new Thread(new DoubleArraySegmentRunner(a,b,first,last,fct));
   threads[i].start();
}
.....
```

Einrichten und Starten jedes Threads sieht so aus wie bisher.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int numberOfThreads = Runtime.getRunTime().availableProcessors() – 1;
int segmentSize = (int) Math.ceil ( (double)a.length / numberOfThreads );
DoubleToDoubleFunction fct = X::someHighlyTimeConsumingFunction;
Thread[ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) {
   int first = i * segmentSize;
   int last = Math.min ( ( i + 1 ) * segmentSize, a.length ) – 1;
   threads[i] = new Thread(new DoubleArraySegmentRunner(a,b,first,last,fct));
   threads[i].start();
}
.....
```

Jetzt sind die Threads gestartet, und wir müssen ihre Terminierung abwarten. Wie das aussieht, sehen wir auf der nächsten Folie.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
boolean allThreadsFinished = false;
while ( ! allThreadsFinished ) {
  allThreadsFinished = true;
  for ( int i = 0; i < threads.length; i++ )
    if ( threads[i].getState() != Thread.State.TERMINATED )
      allThreadsFinished = false;
}
for ( int i = 0; i < b.length; i++ )
  System.out.println ( b[i] );          //  Caveat to follow!
```

Der ursprüngliche Thread beschäftigt sich nach dem Starten der neu erzeugten Threads nur noch damit, reihum abzufragen, ob alle neu erzeugten Threads schon terminiert sind.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
boolean allThreadsFinished = false;
while ( ! allThreadsFinished ) {
  allThreadsFinished = true;
  for ( int i = 0; i < threads.length; i++ )
    if ( threads[i].getState() != Thread.State.TERMINATED )
      allThreadsFinished = false;
}

for ( int i = 0; i < b.length; i++ )
  System.out.println ( b[i] );          //  Caveat to follow!
```

Klasse Thread hat eine Objektmethode namens getState, die den momentanen Status des Threads zum Zeitpunkt des Aufrufs dieser Methode zurückliefert.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
boolean allThreadsFinished = false;
while ( ! allThreadsFinished ) {
  allThreadsFinished = true;
  for ( int i = 0; i < threads.length; i++ )
    if ( threads[i].getState() != Thread.State.TERMINATED )
      allThreadsFinished = false;
}

for ( int i = 0; i < b.length; i++ )
  System.out.println ( b[i] );          //  Caveat to follow!
```

Einer dieser Zustände ist eben, dass der Thread schon terminiert ist.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
boolean allThreadsFinished = false;
while ( ! allThreadsFinished ) {
  allThreadsFinished = true;
  for ( int i = 0; i < threads.length; i++ )
    if ( threads[i].getState() != Thread.State.TERMINATED )
      allThreadsFinished = false;
}
for ( int i = 0; i < b.length; i++ )
  System.out.println ( b[i] );          //  Caveat to follow!
```

Die Konstante gehört zu einem Enum-Typ namens State, der in Klasse Thread eingebettet ist.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
boolean allThreadsFinished = false;
while ( ! allThreadsFinished ) {
  allThreadsFinished = true;
  for ( int i = 0; i < threads.length; i++ )
    if ( threads[i].getState() != Thread.State.TERMINATED )
      allThreadsFinished = false;
}
```

```
for ( int i = 0; i < b.length; i++ )
  System.out.println ( b[i] );
```

```
// Caveat to follow!
```

Wenn alle Threads im Array terminiert sind, dann sind alle Elemente des Arrays b fertig berechnet und können weiterverwendet werden.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Caveat:

- **Das vorangehende Beispiel dient ausschließlich der Illustration von Parallelisierung mit Threads**
- **Es lässt sich aber nicht unbedingt auf andere Situationen übertragen**
- **Dafür gibt es dann (komplexere) Mechanismen in Java**
  - **Suchbegriffe zum Einstieg: Executor{,s,Service}, Callable, Future**

Allerdings muss zum vorangehenden Beispiel eine dringende Warnung ausgesprochen werden.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Caveat:

- **Das vorangehende Beispiel dient ausschließlich der Illustration von Parallelisierung mit Threads**
- **Es lässt sich aber nicht unbedingt auf andere Situationen übertragen**
- **Dafür gibt es dann (komplexere) Mechanismen in Java**
  - **Suchbegriffe zum Einstieg: Executor{,s,Service}, Callable, Future**

Das Beispiel funktioniert so, wie wir es uns angesehen haben, durchaus und illustriert sicherlich gut das Thema Parallelisierung durch Threads.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Caveat:

- **Das vorangehende Beispiel dient ausschließlich der Illustration von Parallelisierung mit Threads**
- **Es lässt sich aber nicht unbedingt auf andere Situationen übertragen**
- **Dafür gibt es dann (komplexere) Mechanismen in Java**
  - **Suchbegriffe zum Einstieg: Executor{,s,Service}, Callable, Future**

Damit es funktioniert, ist das Beispiel auch in ganz bestimmter Weise kreiert worden, nämlich so, dass im Thread kein Objekt neu erstellt und zurückgeliefert, sondern nur ein schon bestehendes Objekt mit Werten gefüllt wird und keiner dieser Werte von zwei verschiedenen Threads überschrieben wird.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Caveat:

- **Das vorangehende Beispiel dient ausschließlich der Illustration von Parallelisierung mit Threads**
- **Es lässt sich aber nicht unbedingt auf andere Situationen übertragen**
- **Dafür gibt es dann (komplexere) Mechanismen in Java**
  - **Suchbegriffe zum Einstieg: Executor{,s,Service}, Callable, Future**

Für den sehr häufigen Fall, dass neue Objekte im erzeugten Thread erstellt und an den erzeugenden Thread zurückgeliefert werden sollen, sind weitere Klassen und Interfaces entwickelt und in die Standardbibliothek aufgenommen worden.

***Nebenbemerkung*: Es wäre bei der Weiterentwicklung von Java auch möglich gewesen, den Kopf von Methode run von Klasse Runnable so abzuändern, dass Runnable auch für komplexere Situationen gereicht hätte. Allerdings war und ist Abwärtskompatibilität ein absolutes Muss bei der Weiterentwicklung einer Progammiersprache. Konkret bedeutet Abwärtskompatibilität hier, dass alter Java-Code, der die Methode run so wie hier gesehen verwendet, nicht geändert werden muss, um weiterhin durch den Compiler zu gehen und zur Laufzeit dasselbe wie bisher zu machen. Daher verbietet sich jede Änderung des Kopfes von run. Überladen der Methode run wäre auch nicht gegangen, denn dann wäre Runnable kein funktionales Interface mehr, weswegen unzählige Programme geändert werden müssten.**

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Caveat:

- **Das vorangehende Beispiel dient ausschließlich der Illustration von Parallelisierung mit Threads**
- **Es lässt sich aber nicht unbedingt auf andere Situationen übertragen**
- **Dafür gibt es dann (komplexere) Mechanismen in Java**
  - **Suchbegriffe zum Einstieg: Executor{,s,Service}, Callable, Future**

Wir werden diese Mechanismen in der FOP *nicht* besprechen, denn uns geht es nur um das Grundverständnis; professionelle Programmierung mit Threads wird in weiterführenden Lehrveranstaltungen thematisiert werden. Hier sehen Sie die Begriffe, nach denen Sie suchen müssen, um auf eigene Faust mehr dazu zu erfahren.

Parallelisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Caveat:

- **Das vorangehende Beispiel dient ausschließlich der Illustration von Parallelisierung mit Threads**
- **Es lässt sich aber nicht unbedingt auf andere Situationen übertragen**
- **Dafür gibt es dann (komplexere) Mechanismen in Java**
  - **Suchbegriffe zum Einstieg: Executor{,s,Service}, Callable, Future**

Ein Wort noch zu dieser Schreibweise: Dieser Regular Expression ist eine kleine Spielerei des Autors dieser Folien; der Ausdruck steht für die Aufzählung von drei verschiedenen Wörtern: Executor, Executors und ExecutorService.

***Erinnerung*: In Kapitel 03b hatten wir Regular Expressions kurz im Zusammenhang mit Strings angesprochen.**

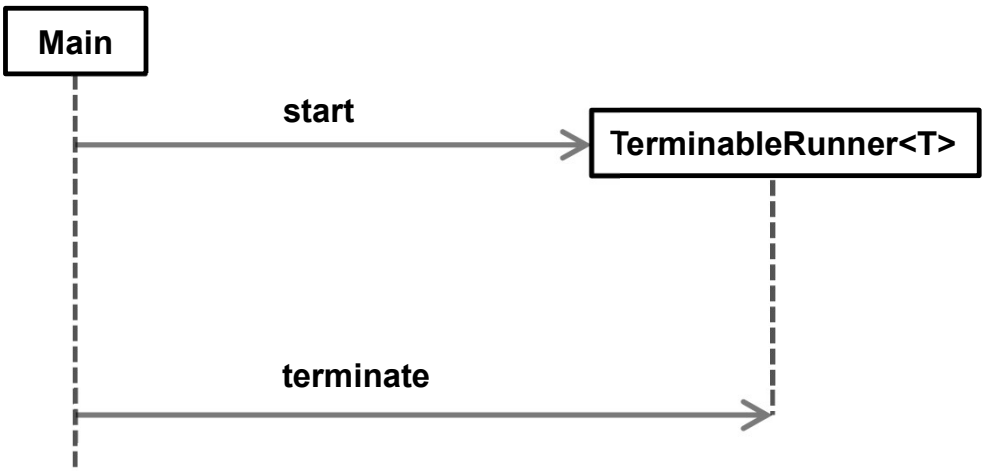
TECHNISCHE UNIVERSITÄT DARMSTADT

Thread terminieren

Bisher hatten wir Threads einfach laufen lassen, bis sie mit ihrer Arbeit fertig sind. In vielen Situationen will man einen einmal erzeugten Thread aber auch wieder beenden, sobald eine bestimmte Bedingung eingetreten ist. Wir schauen uns hier nur *eine* Möglichkeit dafür an, nämlich die, die Oracle empfiehlt für den Fall, dass der erzeugte Thread nicht unverzüglich beendet werden muss, also wenn es ausreicht, dass der erzeugte Thread regelmäßig prüft, ob er sich selbst beenden soll.

Thread terminieren

TECHNISCHE UNIVERSITÄT DARMSTADT



Als Beispiel richten wir eine generische Runnable-Klasse ein, die sukzessive die Elemente eines Streams einliest und verarbeitet und zwischen zwei Elementen jeweils prüft, ob der Thread, in dem diese Runnable-Klasse läuft, terminieren soll. Der generische Typparameter von TerminableRunner ist der des Streams.

Thread terminieren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class TerminableRunner <T> implements Runnable {
    private boolean toBeTerminated;
    private Stream<T> stream;
    private Consumer<T> consumer;
    public TerminableRunner ( Stream<T> stream, Consumer<T> consumer ) {
        toBeTerminated = false;
        this.stream = stream;
        this.consumer = consumer;
    }
    .....
}
```

Nun in Java-Code.

Thread terminieren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class TerminableRunner <T> implements Runnable {
   private boolean toBeTerminated;
   private Stream<T> stream;
   private Consumer<T> consumer;
   public TerminableRunner ( Stream<T> stream, Consumer<T> consumer ) {
      toBeTerminated = false;
      this.stream = stream;
      this.consumer = consumer;
   }
   .....
}
```

In diesem boolean-Attribut wird gespeichert, ob der Thread schon beendet werden soll. Zu Beginn soll das natürlich noch nicht der Fall sein, daher wird toBeTerminated im Konstruktor erst einmal auf false gesetzt.

Thread terminieren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class TerminableRunner <T> implements Runnable {
   private boolean toBeTerminated;
   private Stream<T> stream;
   private Consumer<T> consumer;
   public TerminableRunner ( Stream<T> stream, Consumer<T> consumer ) {
      toBeTerminated = false;
      this.stream = stream;
      this.consumer = consumer;
   }
   .....
}
```

Das, was mit jedem Element des Streams passieren soll, ist in einem Consumer kodiert, der zusammen mit dem Stream im Konstruktor übergeben und in einem Attribut gespeichert wird.

***Erinnerung*: In Kapitel 04c, Abschnitt zu Functional Interfaces und Lambda-Ausdrücken, hatten wir die *nicht*generische Variante von Consumer gesehen; in Kapitel 07, Abschnitt zur eigenen LinkedList-Klasse, dann das hier verwendete *generische* Consumer-Interface.**

Thread terminieren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class TerminableRunner <T> implements Runnable {
    private boolean toBeTerminated;
    private Stream<T> stream;
    private Consumer<T> consumer;
    public TerminableRunner ( Stream<T> stream, Consumer<T> consumer ) {
        toBeTerminated = false;
        this.stream = stream;
        this.consumer = consumer;
    }
    .....
}
```

Es fehlt noch die Methode run; die kommt auf der nächsten Folie. Danach müssen wir dann noch eine Methode hinzufügen, die nicht im Interface Runnable definiert ist und die toBeTerminated auf true setzt.

Thread terminieren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void run() {
    while ( ! toBeTerminated ) {
        Optional<T> opt = stream.findFirst();
        if ( ! opt.isPresent() )
          break;
        consumer.accept ( opt.get() );
    }
}
```

Aber zuerst die Methode run.

Thread terminieren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void run() {
    while ( ! toBeTerminated ) {
        Optional<T> opt = stream.findFirst();
        if ( ! opt.isPresent() )
          break;
        consumer.accept ( opt.get() );
    }
}
```

In jedem Durchlauf dieser while-Schleife wird genau einmal geprüft, ob der Thread terminiert werden soll. Sobald das der Fall ist, ist die while-Schleife beendet. Da nach der while-Schleife nichts mehr in der Methode run kommt, ist der ganze Thread damit beendet.

Thread terminieren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void run() {
    while ( ! toBeTerminated ) {
      Optional<T> opt = stream.findFirst();
      if ( ! opt.isPresent() )
        break;
      consumer.accept ( opt.get() );
    }
}
```

Methode findFirst liefert das erste Element.

***Erinnerung*: Zu Beginn von Kapitel 08 hatten wir die generische Klasse Optional besprochen, da diverse Methoden von Interface Stream das jeweilige Element nicht direkt, sondern eingepackt in einem Optional-Objekt zurückliefern.**

Thread terminieren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void run() {
    while ( ! toBeTerminated ) {
        Optional<T> opt = stream.findFirst();
        if ( ! opt.isPresent() )
          break;
        consumer.accept ( opt.get() );
    }
}
```

Dass der Stream zu Ende ist, wird dadurch angezeigt, dass das zurückgelieferte Optional leer ist. Auch in diesem Fall soll natürlich die while-Schleife und somit der gesamte Thread beendet werden, egal ob toBeTerminated true oder false ist.

Thread terminieren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void run() {
    while ( ! toBeTerminated ) {
        Optional<T> opt = stream.findFirst();
        if ( ! opt.isPresent() )
          break;
        consumer.accept ( opt.get() );
    }
}
```

Wenn es ausreicht, dass toBeTerminated immer nur zwischen zwei Aufrufen von Methode accept des Consumers geprüft wird, dann ist das Ziel, einen terminierbaren Thread einzurichten, mit dieser Implementation von run erreicht. Ansonsten könnte man daran denken, das, was in Methode accept passiert, in einzelne Methoden zu zerlegen, so dass toBeTerminated auch dazwischen geprüft werden kann.

Thread terminieren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class TerminableRunner <T> implements Runnable {
  .....
  public void terminate() {
     toBeTerminated = true;
  }
}
```

Die letzte noch ausstehende Methode von TerminableRunner soll das Terminieren von außen steuern. Diese ist nun denkbar einfach und bedarf wohl keines Kommentars.

Thread terminieren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Runnable runnable = new TerminableRunner ( stream, consumer );
new Thread(runnable).start();
Thread.sleep ( 200 );
runnable.terminate();
```

Nun ist Klasse TerminableRunner vollständig definiert und kann eingesetzt werden.

Thread terminieren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Runnable runnable = new TerminableRunner ( stream, consumer );
new Thread(runnable).start();
Thread.sleep ( 200 );
runnable.terminate();
```

Wir richten also ein TerminableRunner-Objekt mit einem Stream und einem Consumer ein. Wo die beiden aktualen Parameter des Konstruktors herkommen, interessiert hier nicht.

Thread terminieren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Runnable runnable = new TerminableRunner ( stream, consumer );
new Thread(runnable).start();
Thread.sleep ( 200 );
runnable.terminate();
```

Dann starten wir den Thread und vergessen ihn wieder. Denn nicht den Thread selbst, sondern das Runnable-Objekt wollen wir später noch einmal ansprechen.

Thread terminieren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Runnable runnable = new TerminableRunner ( stream, consumer );
new Thread(runnable).start();
Thread.sleep ( 200 );
runnable.terminate();
```

Der erzeugende Thread macht jetzt mit seinen eigenen Aufgaben weiter, bis er den Punkt erreicht hat, an dem er den soeben erzeugten Thread terminieren möchte. In diesem illustrativen Beispiel machen wir es uns einfach: Nach zweihundert Millisekunden ist die Terminierungsbedingung erfüllt, und bis dahin hat der erzeugende Thread nichts zu tun und kann sich schlafen legen.

Thread terminieren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Runnable runnable = new TerminableRunner ( stream, consumer );
new Thread(runnable).start();
Thread.sleep ( 200 );
runnable.terminate();
```

Und nach diesen zweihundert Millisekunden wird der TerminableRunner in diesem Thread benachrichtigt, so dass er die while-Schleife nach dem nächsten Durchlauf beendet, wodurch die Methode run und daher auch der Thread beim nächsten Beginn der while-Schleife in terminate beendet werden.

Damit ist dieses Beispiel zur Terminierung von Threads aus anderen Threads heraus abgeschlossen.

TECHNISCHE UNIVERSITÄT DARMSTADT

Wozu Threads?

Nach den diversen Fallbeispielen können wir jetzt daran gehen, die auf dieser Folie aufgeworfene Frage umfassend zu beantworten.

Wozu Threads?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Parallelisierung (auch remote)
  - Achtung: nicht immer schneller
  - Man muss genau wissen, was man tut
- Abspaltung von eigenständig arbeitenden Programmteilen
  - Starten und vergessen („fire and forget“)
  - Starten und später nochmals ansprechen

Es sind im Wesentlichen zwei Gründe, warum man gerne Threads verwenden möchte.

Wozu Threads?

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Parallelisierung (auch remote)**
  - **Achtung: nicht immer schneller**
  - **Man muss genau wissen, was man tut**
- **Abspaltung von eigenständig arbeitenden Programmteilen**
  - **Starten und vergessen („fire and forget“)**
  - **Starten und später nochmals ansprechen**

***Einen* Grund haben wir *explizit* diskutiert: Beschleunigung eines zeitintensiven Vorgangs durch Aufteilung auf mehrere Prozessoren.**

Wozu Threads?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Parallelisierung (auch remote)
  - Achtung: nicht immer schneller
  - Man muss genau wissen, was man tut
- Abspaltung von eigenständig arbeitenden Programmteilen
  - Starten und vergessen („fire and forget“)
  - Starten und später nochmals ansprechen

Darunter fällt auch die Möglichkeit, Threads remote, also auf anderen Computern zu starten, so dass sich also mehrere Computer die Arbeitslast teilen. Das ist aber definitiv nicht mehr Gegenstand der FOP, daher wurde dieser Aspekt hier ausgelassen.

Wozu Threads?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Parallelisierung (auch remote)
  - Achtung: nicht immer schneller
  - Man muss genau wissen, was man tut
- Abspaltung von eigenständig arbeitenden Programmteilen
  - Starten und vergessen („fire and forget“)
  - Starten und später nochmals ansprechen

Wenn man versucht, die Laufzeit eines Programms zu verbessern, hat man es häufig mit Phänomenen zu tun, die man nicht mehr durchschauen kann, so dass unangenehme Überraschungen keine Seltenheit sind: Man hat kluge Überlegungen angestellt und diese mit viel Zeit und Mühe implementiert – und das Programm ist keinen Deut schneller, vielleicht sogar merklich langsamer geworden!

Parallelisierung mit Hilfe von Threads ist ein notorisches Beispiel dafür.

Wozu Threads?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Parallelisierung (auch remote)
  - Achtung: nicht immer schneller
  - Man muss genau wissen, was man tut
- Abspaltung von eigenständig arbeitenden Programmteilen
  - Starten und vergessen („fire and forget“)
  - Starten und später nochmals ansprechen

Daher sollte man besser nicht in‘s Blaue hinein versuchen, Laufzeiten durch Parallelisierung zu verbessern, sondern zumindest vorher die Lehrveranstaltungen zum Thema Parallelisierung besucht haben.

Wozu Threads?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Parallelisierung (auch remote)
  - Achtung: nicht immer schneller
  - Man muss genau wissen, was man tut
- Abspaltung von eigenständig arbeitenden Programmteilen
  - Starten und vergessen („fire and forget“)
  - Starten und später nochmals ansprechen

Die meisten unserer bisherigen Beispiele haben allerdings eher *diesen* Punkt beleuchtet. Das ist auch der Aspekt, unter dem wir Threads in Kapitels 10 sehen werden, wenn es um GUIs geht.

Wozu Threads?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Parallelisierung (auch remote)
  - Achtung: nicht immer schneller
  - Man muss genau wissen, was man tut
- Abspaltung von eigenständig arbeitenden Programmteilen
  - Starten und vergessen („fire and forget“)
  - Starten und später nochmals ansprechen

Im Wesentlichen waren unsere Beispiele bisher von diesem Typ, wir haben meist nicht einmal das Ergebnis von Operator new für spätere Verwendung gespeichert.

Wozu Threads?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Parallelisierung (auch remote)
  - Achtung: nicht immer schneller
  - Man muss genau wissen, was man tut
- Abspaltung von eigenständig arbeitenden Programmteilen
  - Starten und vergessen („fire and forget“)
  - Starten und später nochmals ansprechen

Das letzte Beispiel war von der zweiten Art: Wir haben den Thread später noch einmal angesprochen, in diesem Beispiel einfach nur um ihn zu terminieren.

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallele Streams

Threads sind in Streams schon eingebaut, so dass wir uns darum nicht kümmern müssen.

Parallele Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double averageAgeOfDarmstadtResidents
   = list.parallelStream()
         .filter ( p -> p.residency == “Darmstadt“ )
         .map ( p -> p.age )
         .average()
         .getAsDouble();
```

.

Hier sehen Sie ein typisches Beispiel für die Arbeit mit Streams, ähnlich denen, die wir früher schon gesehen hatten.

***Erinnerung*: Ähnliche Beispiele haben Sie im Abschnitt zu Streams in Kapitel 08 ausführlich gesehen, so dass hier keine weiteren Erläuterungen notwendig sein sollten.**

Parallele Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double averageAgeOfDarmstadtResidents
   = list.parallelStream()
         .filter ( p -> p.residency == “Darmstadt“ )
         .map ( p -> p.age )
         .average()
         .getAsDouble();
```

.

Der Unterschied ist, dass die Methode stream von Collection ersetzt ist durch die Methode parallelStream. Die Methode parallelStream liefert ebenfalls einen Stream zurück, der alle Listenelemente enthält und auch ebenfalls in derselben Reihenfolge wie in der Liste. Diese Methode kann, muss aber nicht eine Aufteilung des Streams auf mehrere Threads realisieren, diese Entscheidung wird in Methode parallelStream getroffen. Die weiteren Methodenaufrufe in dieser Anweisung arbeiten dann ebenfalls parallel in diesen Threads.

Offenkundig ist das eine sehr bequeme Möglichkeit, um die Verarbeitung großer Datenmengen zu parallelisieren. Und mutmaßlich auch eine besonders effiziente, denn man kann mit einigem Recht davon ausgehen, dass die Methode parallelStream eine sehr gute Entscheidung treffen wird, in wie viele Threads der Stream zerlegt werden soll.