- Jedes Objekt (mit `synchronized`-Methoden) verfügt über eine weitere Schlange `ThreadQueue waitingThreads` am Objekt wartender Threads sowie die Objekt-Methoden:

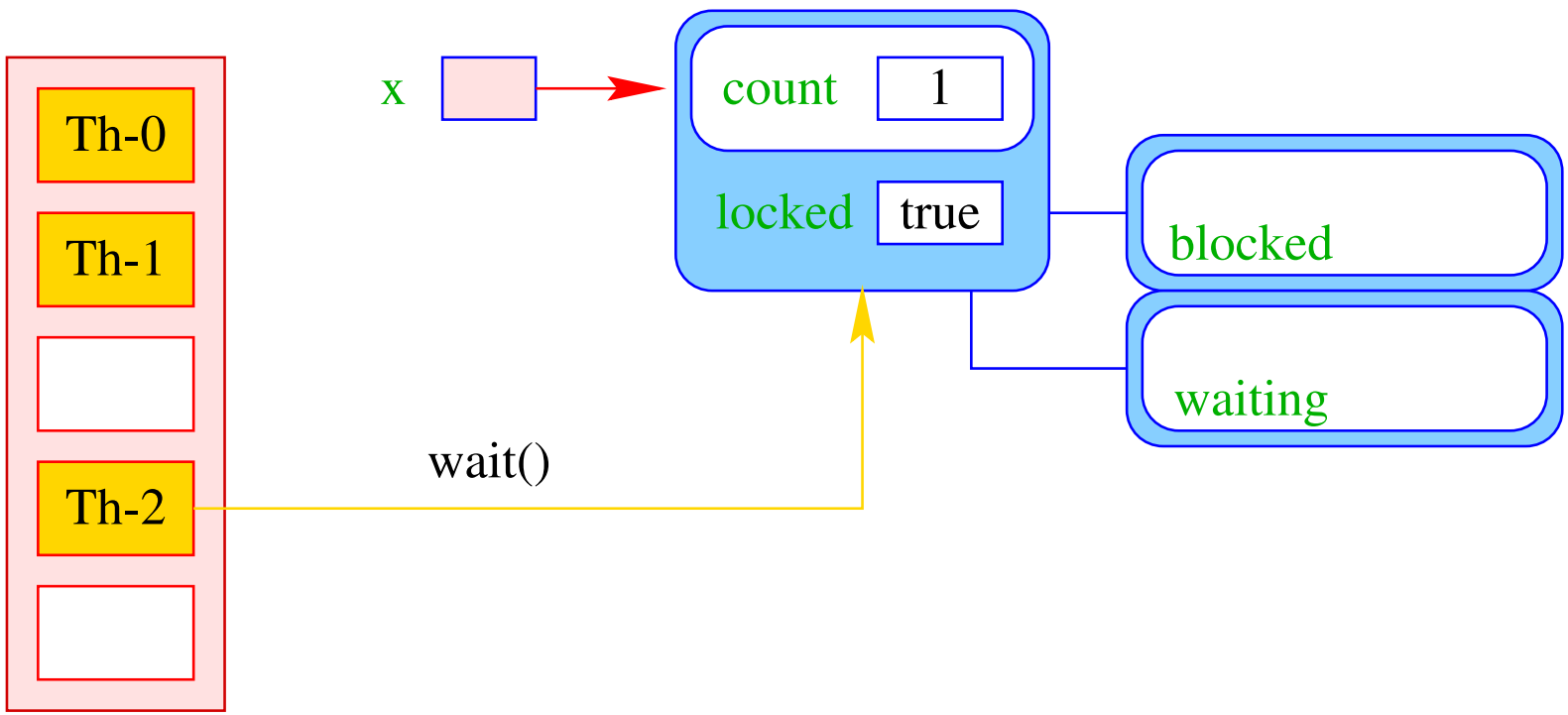
  ```
  public final void wait() throws InterruptedException;
  ```

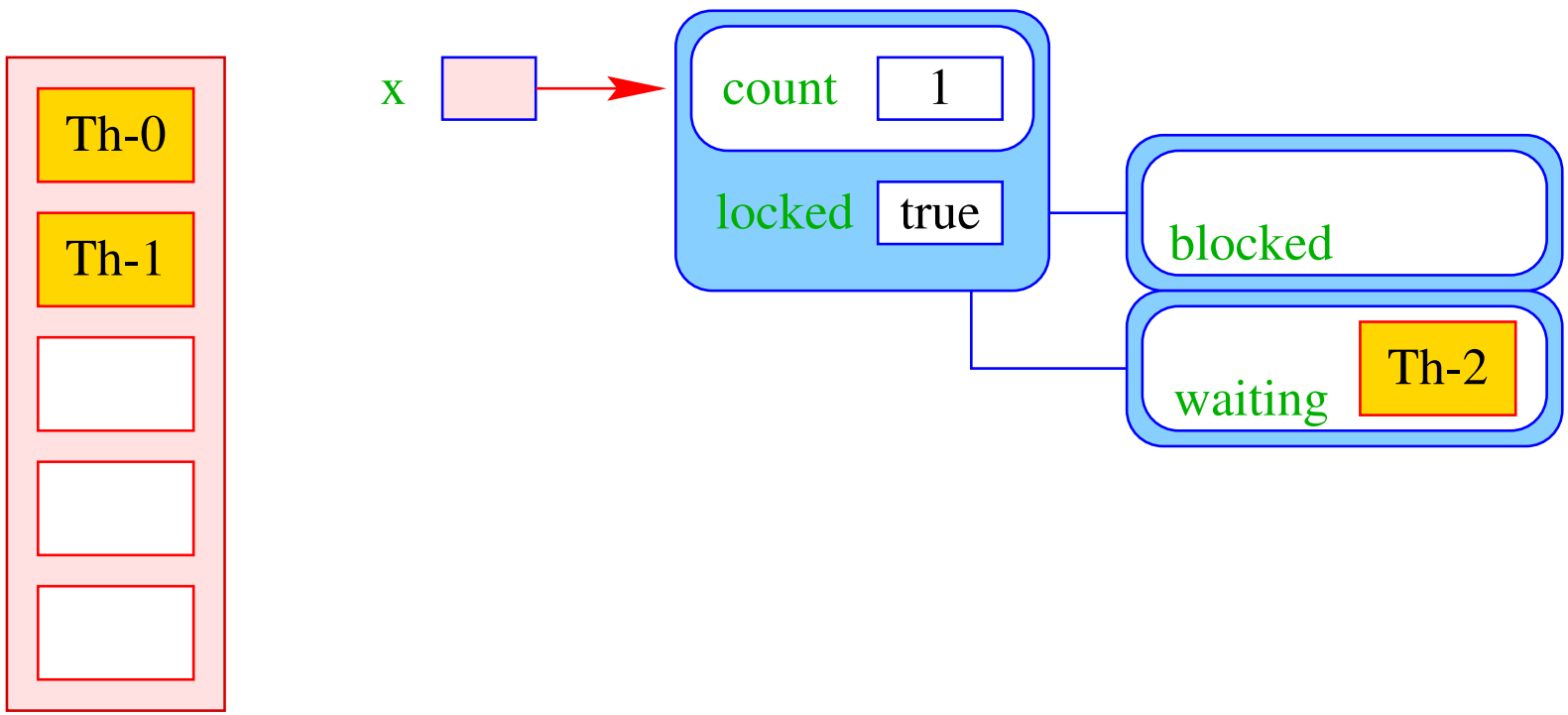
  ```
  public final void notify();
  ```

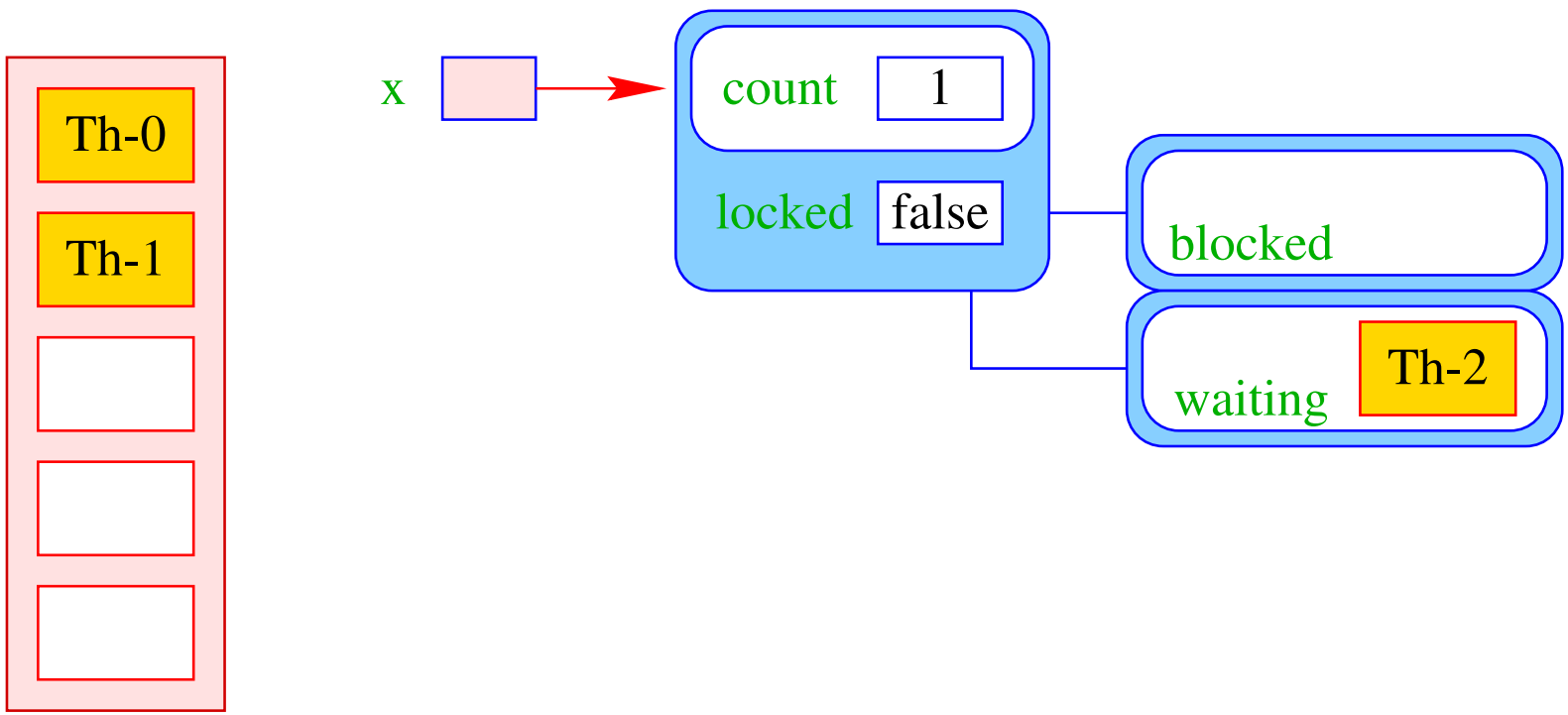
  ```
  public final void notifyAll();
  ```

- Diese Methoden dürfen nur für Objekte aufgerufen werden, über deren Lock der Thread verfügt !!!
- Ausführen von `wait();` setzt den Zustand des Threads auf waiting, reiht ihn in eine geeignete Warteschlange ein, und gibt das aktuelle Lock frei:

```
public void wait() throws InterruptedException {
   Thread t = Thread.currentThread();
   t.state = waiting;
   waitingThreads.enqueue(t);
   unlock();
}
```


Th-0
Th-1
Th-2
x
count
1
locked
true
blocked
waiting
wait()


Th-0
Th-1
x
count
1
locked
true
blocked
waiting
Th-2


Th-0
Th-1
x
count
1
locked
false
blocked
waiting
Th-2

- Ausführen von `notify();` weckt den ersten Thread in der Warteschlange auf, d.h. versetzt ihn in den Zustand ready ...

```
public void notify() {
   if (!waitingThreads.isEmpty()) {
      Thread t = waitingThreads.dequeue();
      t.state = ready;
   }
}
```

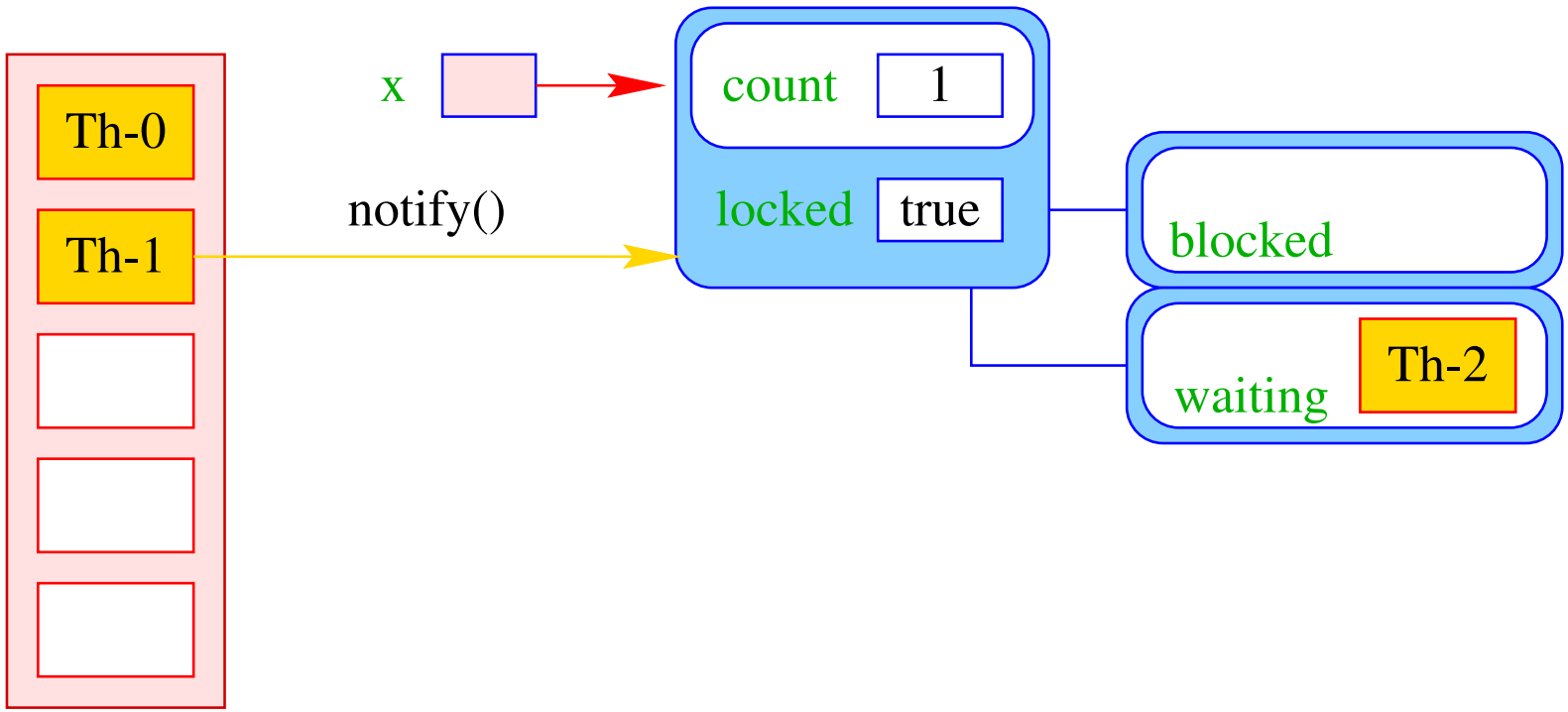
- ... mit der Auflage, erneut das Lock zu erwerben, d.h. als erste Operation hinter dem `wait();` ein `lock()` auszuführen.

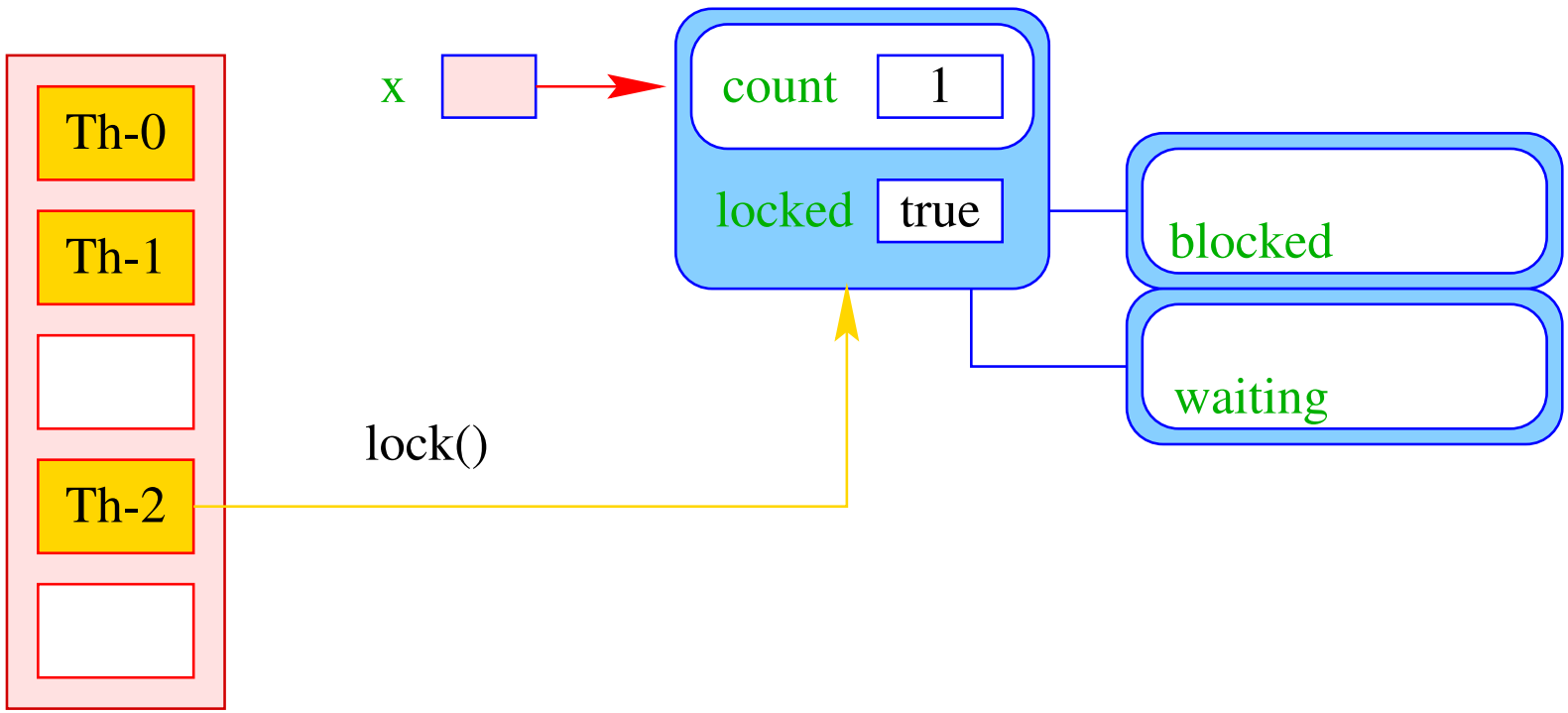
- Ausführen von `notify();` weckt den ersten Thread in der Warteschlange auf, d.h. versetzt ihn in den Zustand ready ...

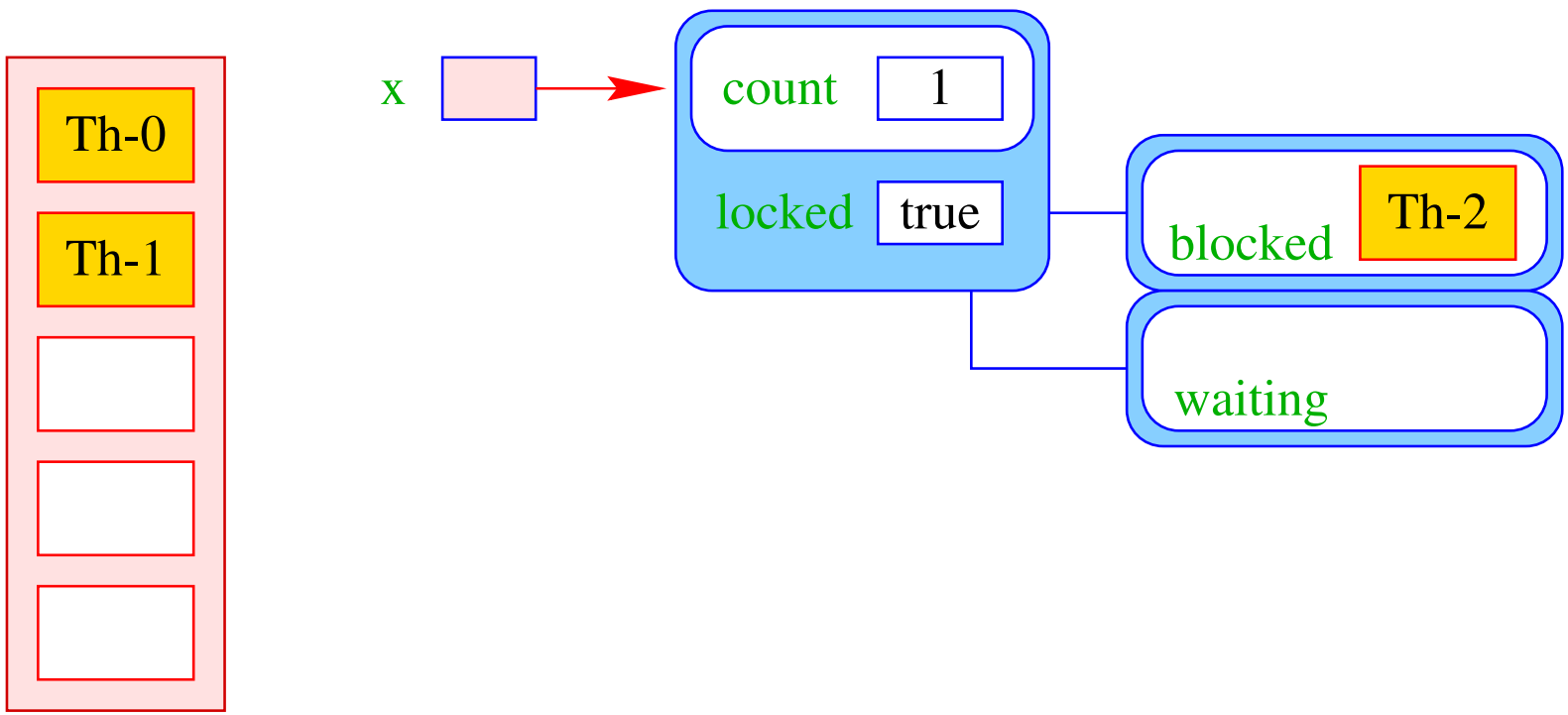
```
public void notify() {
   if (!waitingThreads.isEmpty()) {
      Thread t = waitingThreads.dequeue();
      t.state = ready;
   }
}
```

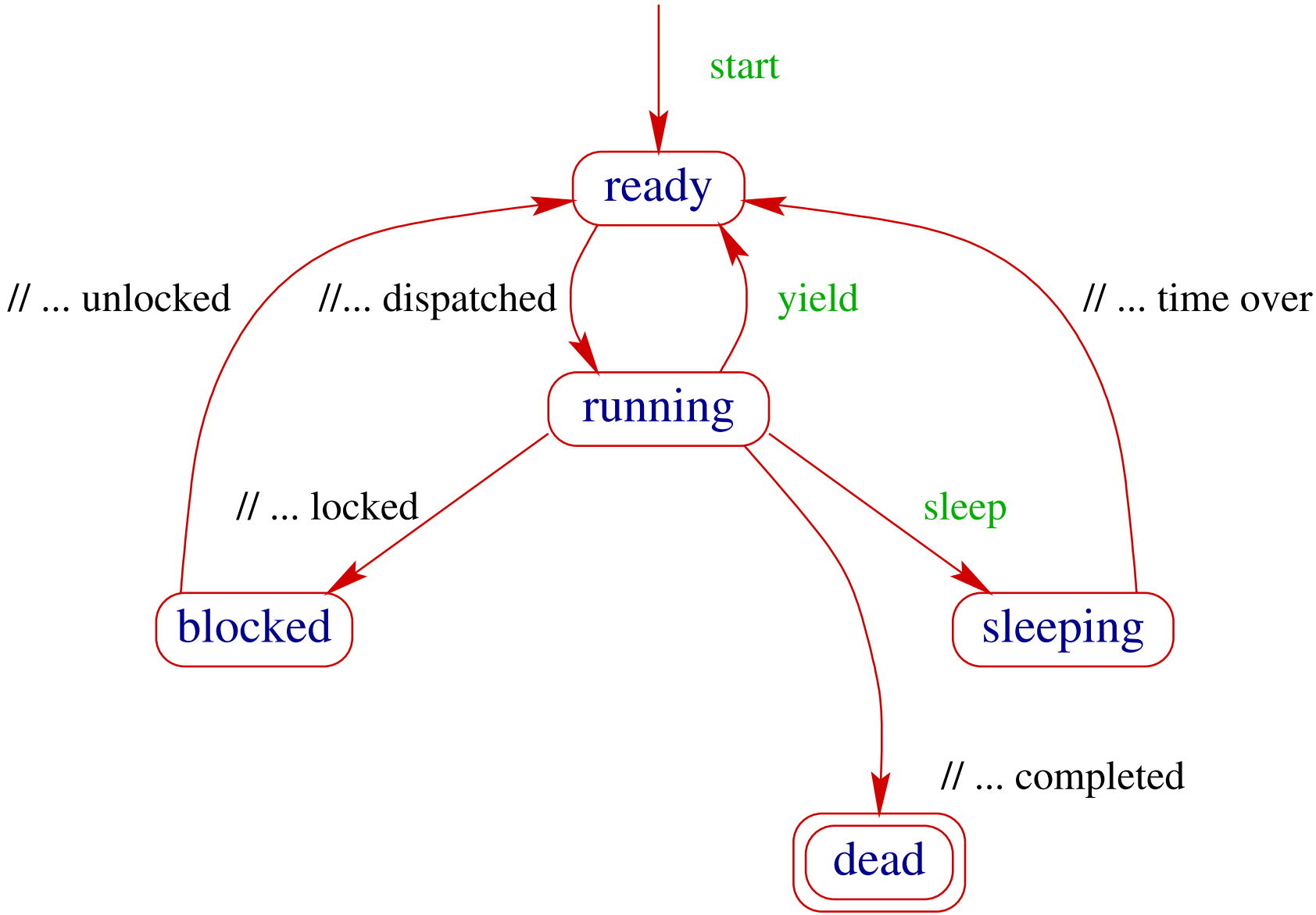
- ... mit der Auflage, erneut das Lock zu erwerben, d.h. als erste Operation hinter dem `wait();` ein `lock()` auszuführen.
- `notifyAll();` weckt alle wartenden Threads auf:

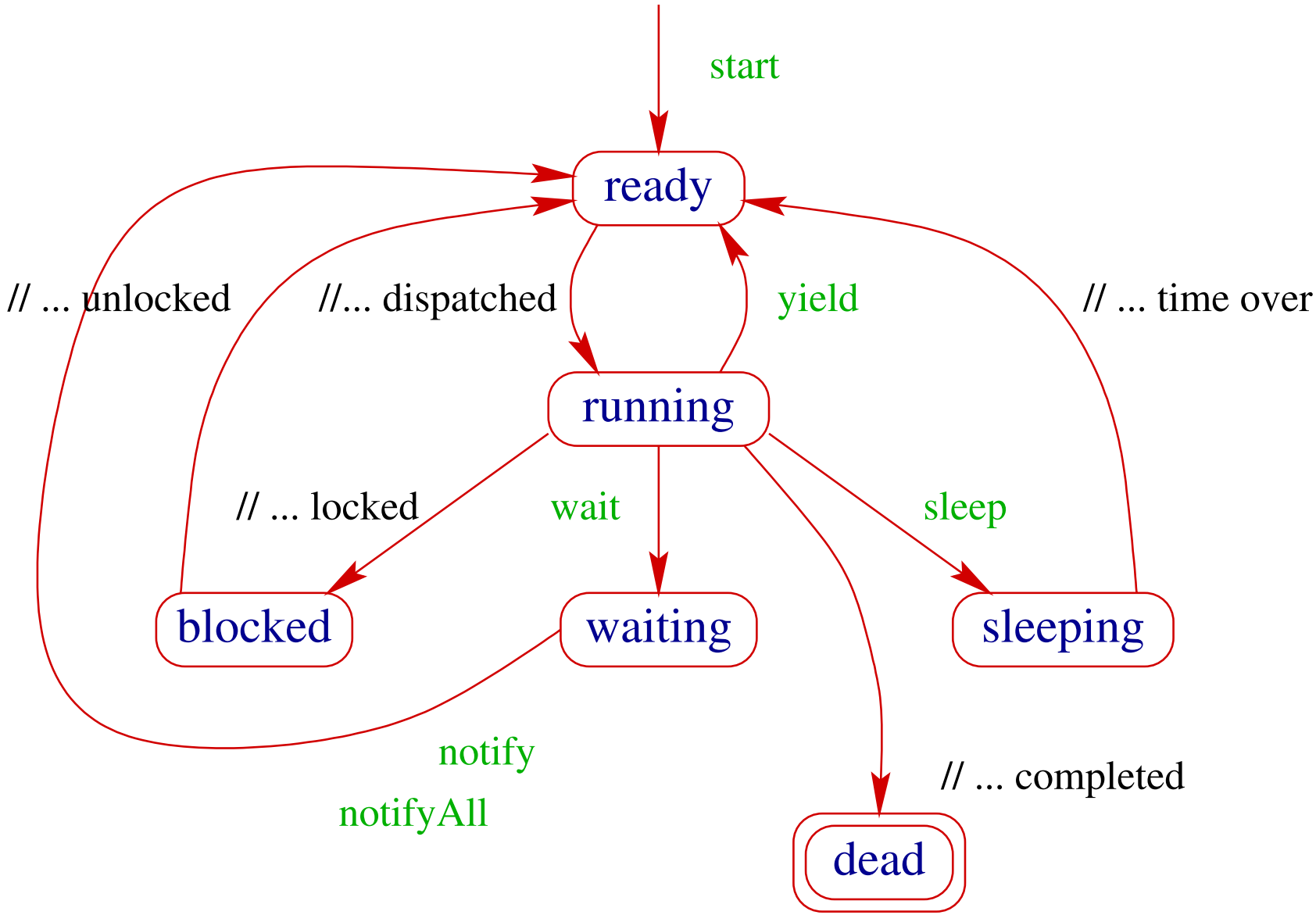
```
public void notifyAll() {
   while (!waitingThreads.isEmpty()) notify();
}
```


Th-0
Th-1
x
notify()
count
1
locked
true
blocked
waiting
Th-2


Th-0
Th-1
Th-2
x
count
1
locked
true
blocked
waiting
lock()


Th-0
Th-1
x
count
1
locked
true
blocked
Th-2
waiting


start
ready
// ... unlocked
//... dispatched
yield
// ... time over
running
// ... locked
sleep
blocked
sleeping
// ... completed
dead


start
ready
// ... unlocked
//... dispatched
yield
// ... time over
running
// ... locked
wait
sleep
blocked
waiting
sleeping
notify
notifyAll
// ... completed
dead

Anwendung:

```
   ...
   public synchronized void produce(Data d) throws InterruptedException {
      if (free==0) wait(); free--;
      a[last] = d;
      last = (last+1)%cap;
      notify();
   }
   public synchronized Data consume() throws InterruptedException {
      if (free==cap) wait(); free++;
      Data result = a[first];
      first = (first+1)%cap;
      notify(); return result;
   }
}  // end of class Buffer2
```

- Ist der Puffer voll, d.h. keine Zelle frei, legt sich der Producer schlafen.
- Ist der Puffer leer, d.h. alle Zellen frei, legt sich der Consumer schlafen.
- Gibt es für einen Puffer genau einen Producer und einen Consumer, weckt das `notify()` des Consumers (wenn überhaupt, dann) stets den Producer ...

  ... und umgekehrt.
- Was aber, wenn es mehrere Producers gibt? Oder mehrere Consumers ???

2. Idee: Wiederholung der Tests

- Teste nach dem Aufwecken erneut, ob Zellen frei sind.
- Wecke nicht einen, sondern alle wartenden Threads auf ...

```
...
public synchronized void produce(Data d)
                                 throws InterruptedException {
   while (free==0) wait(); free--;
   a[last] = d;
   last = (last+1)%cap;
   notifyAll();
}
...
```

```
    ...
    public synchronized Data consume() throws InterruptedException {
        while (free==cap) wait();
        free++;
        Data result = a[first];
        first = (first+1)%cap;
        notifyAll();
        return result;
    }
}   // end of class Buffer2
```

- Wenn ein Platz im Puffer frei wird, werden sämtliche Threads aufgeweckt – obwohl evt. nur einer der Producer bzw. nur einer der Consumer aktiv werden kann :-(

3. Idee: Semaphore

- Producers und Consumers warten in verschiedenen Schlangen.
- Die Producers warten darauf, dass `free > 0` ist.
- Die Consumers warten darauf, dass `cap-free > 0` ist.

3. Idee: Semaphore

- Producers und Consumers warten in verschiedenen Schlangen.
- Die Producers warten darauf, dass `free > 0` ist.
- Die Consumers warten darauf, dass `cap-free > 0` ist.

```
public class Sema { private int x;
   public Sema(int n) { x = n; }
   public synchronized void up() {
      x++; if (x<=0) notify();
   }
   public synchronized void down() throws InterruptedException {
      x--; if (x<0) wait();
   }
}  // end of class Sema
```

- Ein Semaphor enthält eine private `int`-Objekt-Variable und bietet die `synchronized`-Methoden `up()` und `down()` an.
- `up()` erhöht die Variable, `down()` erniedrigt sie.
- Ist die Variable positiv, gibt sie die Anzahl der verfügbaren Resourcen an.

  Ist sie negativ, zählt sie die Anzahl der wartenden Threads.
- Eine `up()`-Operation weckt genau einen wartenden Thread auf.

Anwendung (1. Versuch :-)

```
public class Buffer {
   private int cap, first, last;
   private Sema free, occupied;
   private Data[] a;
   public Buffer(int n) {
      cap = n; first = last = 0;
      a = new Data[n];
      free = new Sema(n);
      occupied = new Sema(0);
   }
   ...
```

```
   ...
   public synchronized void produce(Data d) throws InterruptedException {
      free.down();
      a[last] = d;
      last = (last+1)%cap;
      occupied.up();
   }
   public synchronized Data consume() throws InterruptedException {
      occupied.down();
      Data result = a[first];
      first = (first+1)%cap;
      free.up();
      return result;
   }
}  // end of faulty class Buffer
```

- Gut gemeint – aber leider fehlerhaft ...
- Jeder Producer benötigt zwei Locks gleichzeitig, um zu produzieren:
  1. dasjenige für den Puffer;
  2. dasjenige für einen Semaphor.
- Muss er für den Semaphor ein `wait()` ausführen, gibt er das Lock für den Semaphor wieder zurück ... nicht aber dasjenige für den Puffer !!!
- Die Folge ist, dass niemand mehr eine Puffer-Operation ausführen kann, insbesondere auch kein `up()` mehr für den Semaphor $\Longrightarrow$ Deadlock

- Gut gemeint – aber leider fehlerhaft ...
- Jeder Producer benötigt zwei Locks gleichzeitig, um zu produzieren:
  1. dasjenige für den Puffer;
  2. dasjenige für einen Semaphor.
- Muss er für den Semaphor ein `wait()` ausführen, gibt er das Lock für den Semaphor wieder zurück ... nicht aber dasjenige für den Puffer !!!
- Die Folge ist, dass niemand mehr eine Puffer-Operation ausführen kann, insbesondere auch kein `up()` mehr für den Semaphor $\Longrightarrow$ Deadlock

Anwendung (2. Versuch :-) Entkopplung der Locks

```
    ...
    public void produce(Data d) throws InterruptedException {
       free.down();
       synchronized (this) {
          a[last] = d; last = (last+1)%cap;
       }
       occupied.up();
    }
    public Data consume() throws InterruptedException {
       occupied.down();
       synchronized (this) {
          Data result = a[first]; first = (first+1)%cap;
       }
       free.up(); return result;
    }
}   // end of corrected class Buffer
```

- Das Statement `synchronized ( obj ) { stmts }` definiert einen kritischen Bereich für das Objekt obj, in dem die Statement-Folge stmts ausgeführt werden soll.
- Threads, die die neuen Objekt-Methoden `void produce(Data d)` bzw. `Data consume()` ausführen, benötigen zu jedem Zeitpunkt nur genau ein Lock :-)

Warnung:

Threads sind nützlich, sollten aber nur mit Vorsicht eingesetzt werden. Es ist besser,

- ... wenige Threads zu erzeugen als mehr.
- ... unabhängige Threads zu erzeugen als sich wechselseitig beeinflussende.
- ... kritische Abschnitte zu schützen, als nicht synchronisierte Operationen zu erlauben.
- ... kritsche Abschnitte zu entkoppeln, als sie zu schachteln.

Warnung (Forts.):

Finden der Fehler bzw. Überprüfung der Korrektheit ist ungleich schwieriger als für sequentielle Programme:

- Fehlerhaftes Verhalten tritt eventuell nur gelegentlich auf...
- bzw. nur für bestimmte Scheduler :-(
- Die Anzahl möglicher Programm-Ausführungsfolgen mit potentiell unterschiedlichem Verhalten ist gigantisch.

22 Applets

Applet $\quad\Longleftrightarrow\quad$ Java-Programm in einer HTML-Seite

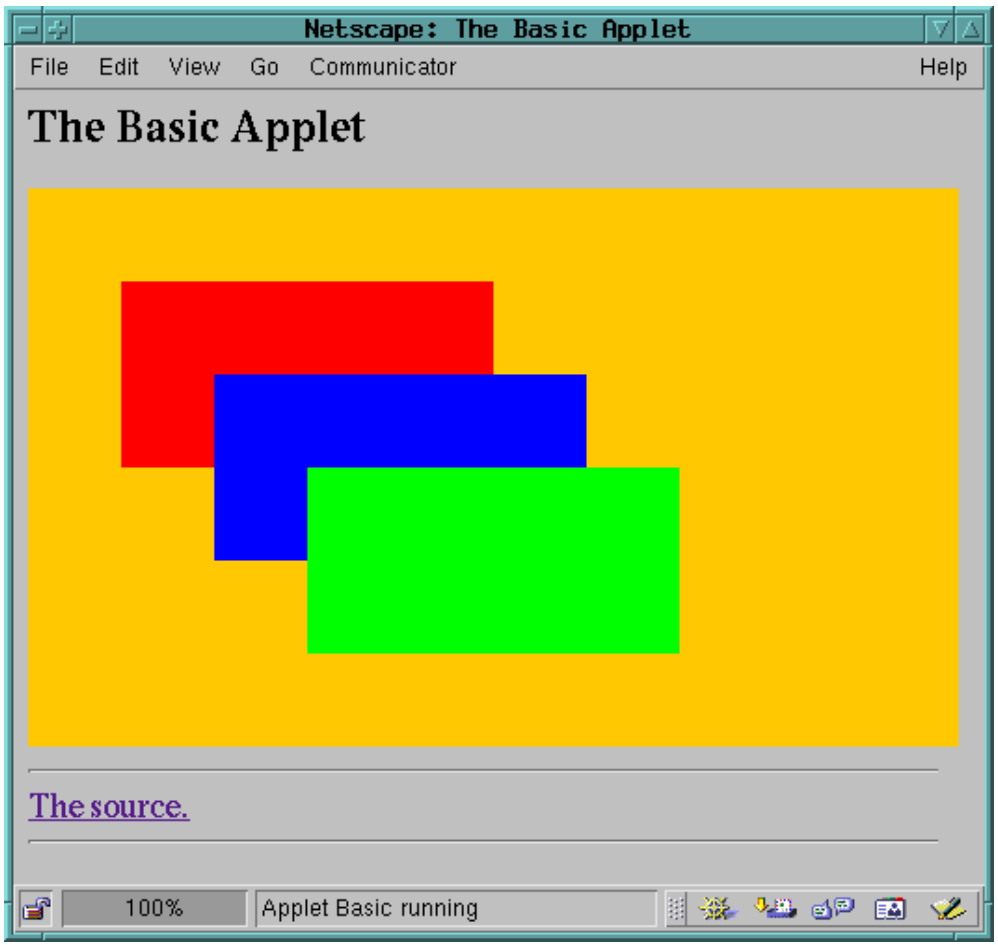
Ziele:

- audio-visuelle Gestaltung der Seite;
- Animation, d.h. Darstellen von Abfolgen von Bildern (evt. parallel zur Wiedergabe von Ton)
- Interaktion :-)

Achtung:

- Verfügbarkeit toller technischer Möglichkeiten bedeutet noch lange nicht, dass das Ergebnis beeindruckend ist ...
  ↑Kreativität
- Applets werden aus dem Internet gezogen:
  - Es gibt keine Garantie, dass sie nicht bösartig sind, d.h. versuchen, unerwünschte Effekte hervorzurufen ...
  - Applets werden darum innerhalb des Browsers ausgeführt und haben nur begrenzten Zugang zu System-Resourcen. ↑Sicherheit
- Statt im Browser können Applets auch mithilfe des Programms `appletviewer` betrachtet werden.

Ein Beispiel:



```
<html><head><title>The Basic Applet </title>
            <!-- Created by: Helmut Seidl -->
    </head>
    <body><h1>The Basic Applet</h1>
    <applet   codebase="."
              documentbase="pictures"
              code=Basic.class
              width=500
              height=300>
    Your browser is completely ignoring the &lt;APPLET&gt; tag!
    </applet>
    <hr>
    <a href="Basic.java">The source.</a>
    <hr>
    </body>
</html>
```

- HTML-Seiten bestehen aus (möglicherweise geschachtelten) Elementen, Kommentaren und Text.
- Ein Kommentar beginnt mit `<!--` und endet mit `-->`.
- Nicht-leere Elemente `b` beginnen mit einem Start-Tag `<b>` und enden mit einem End-Tag `</b>`.

Beispiele:

`html`	—	die gesamte Seite;
`head`	—	der Kopf der Seite;
`title`	—	der Titel der Seite;
`body`	—	der Rumpf der Seite;
`a`	—	ein Link;
`h1`	—	eine Überschrift;
`applet`	—	ein Applet.

- Bei leeren Elementen kann das End-Tag auch fehlen.

Beispiele:

`br` — Zeilen-Umbruch;

`hr` — horizontale Linie.

- Sonderzeichen beginnen mit `&` und enden mit `;` (Character references).

Beispiele:

`<` — “<”

`>` — “>”

`ä` — “ä”