- 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 :-)

- 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 :-)

20.3 Interrupts

Problem:

- Zeit ist kostbar.
- Gerne möchte man nach einiger Zeit das Warten abbrechen ...
- oder einem Thread mitteilen, dass er nicht länger warten soll.

- Analog zur Klassen-Methode `public void sleep(int msec)` der Klasse `Thread` gibt es die Objekt-Methoden

  ```
  public void wait(int msec);
  public void wait(int msec, int nsec);
  ```

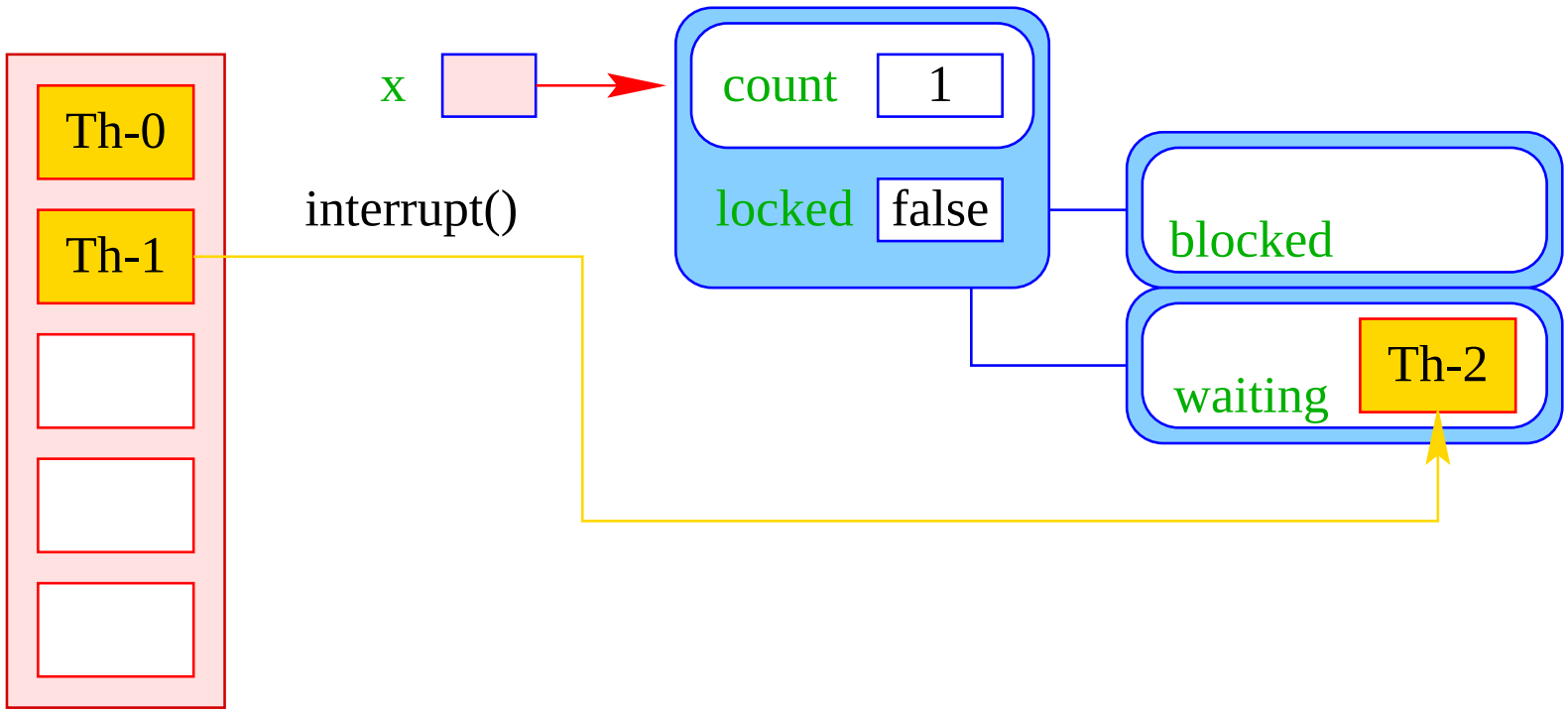
  der Klasse `Object`, die das Warten auf `msec` Millisekunden bzw. `msec` Millisekunden und `nsec` Nanosekunden begrenzen.
- Jedem Thread kann ein Interrupt-Signal gesendet werden.
- Jeder Thread verfügt über ein Flag `boolean interrupted`, das anzeigt, ob er ein Interrupt-Signal erhielt.
- Die Objekt-Methode `public void interrupt();` der Klasse `Thread` sendet einem `Thread`-Objekt ein Interrupt-Signal mit den folgenden Effekt:

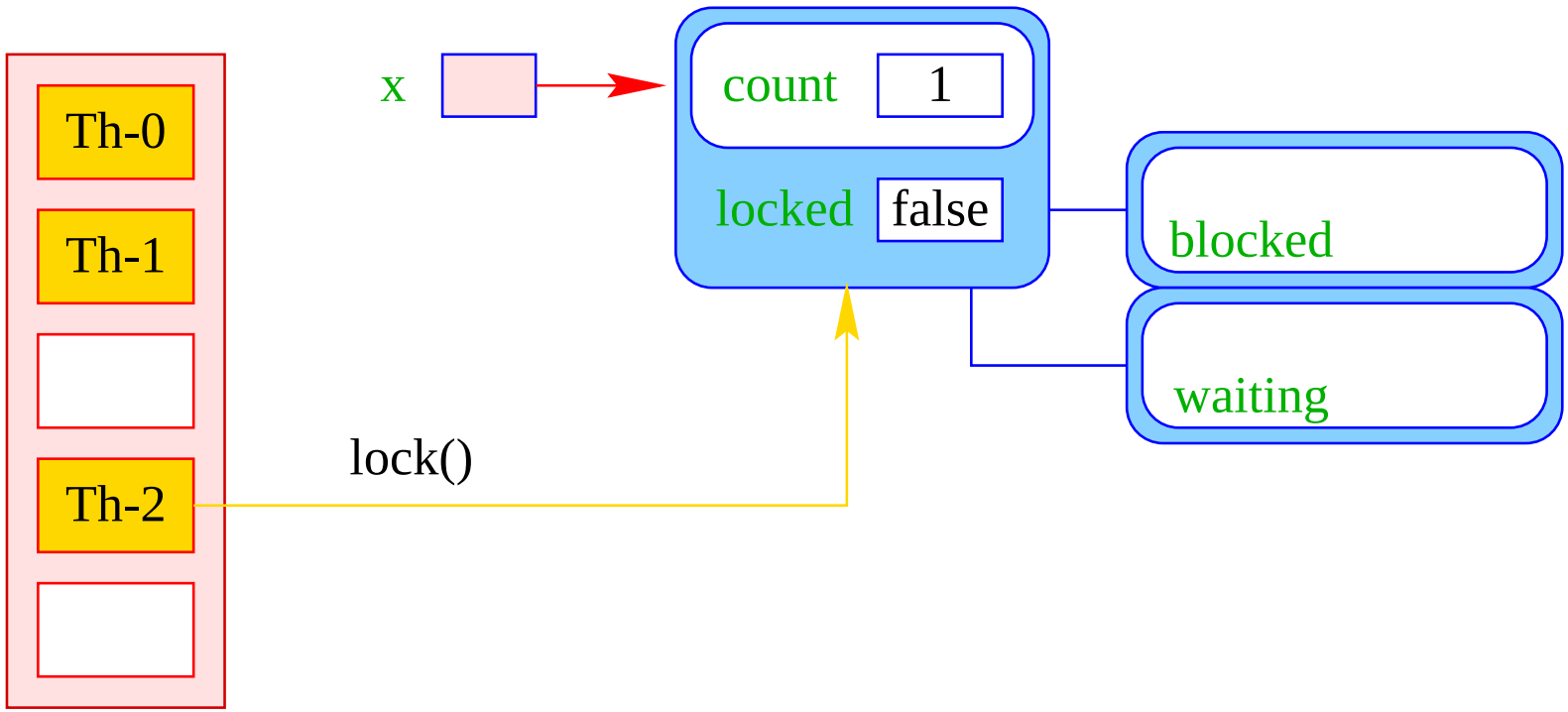
  1. Das Flag `interrupted` wird gesetzt;
  2. der Zustand des Threads wird auf ready gesetzt – sofern er nicht running oder blocked ist;

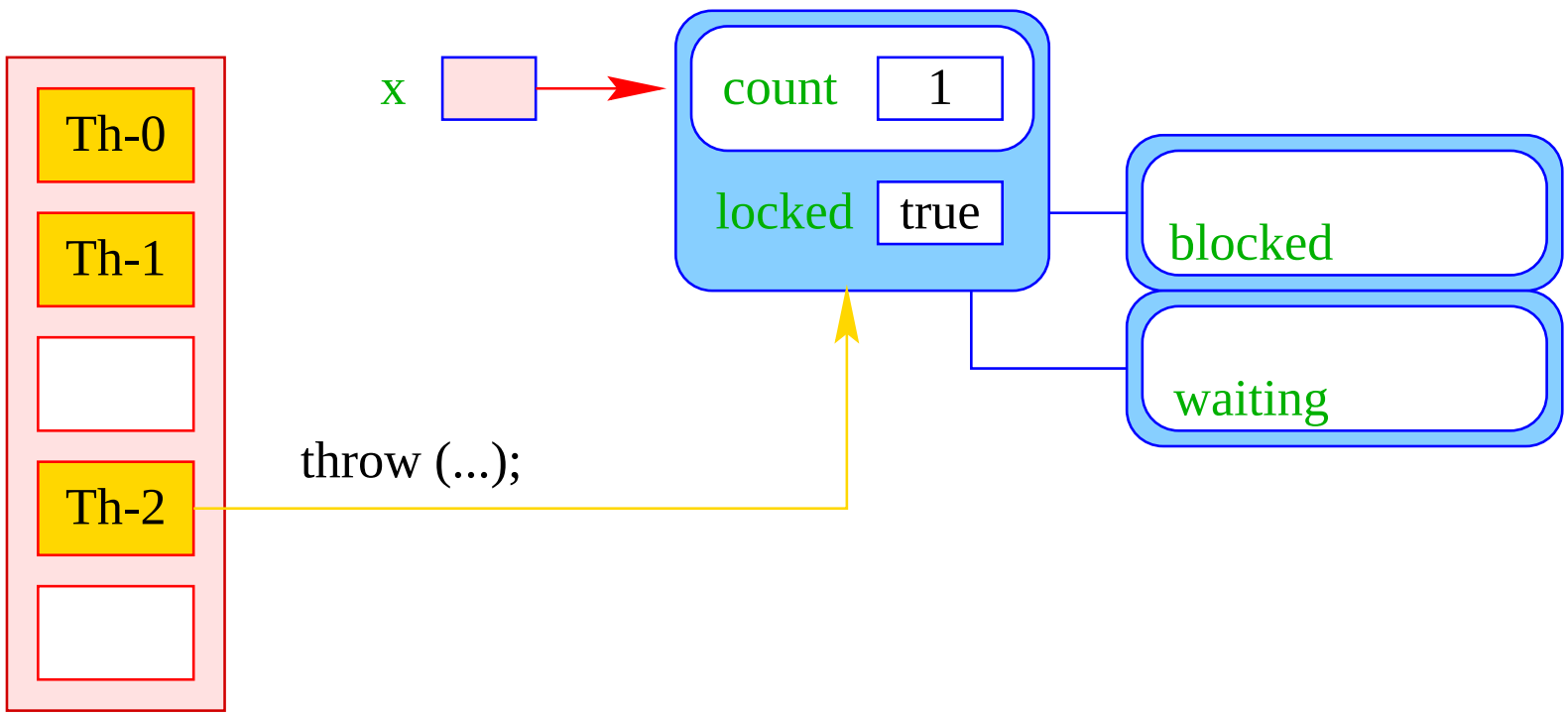
3. Wartete der Thread vorher (z.B. auf die Beendigung eines anderen Thread oder ein `notify()`), wird er aus der Warteschlange entfernt.

- In Abhängigkeit vom vorherigen Zustand `oldState` führt der reaktivierte Thread die folgenden Aktionen aus:

```
switch(oldState) {
   case waiting:   lock();
   case sleeping:
   case joining:   throw (new
        InterruptedException ("operation interrupted"));
   case joiningIO: throw (new
        InterruptedIOException ());
}
```


Th-0
Th-1
x
interrupt()
count
1
locked
false
blocked
waiting
Th-2


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


Th-0
Th-1
Th-2
x
count
1
locked
true
blocked
waiting
throw (...);

Beispiel:

```
public class Simple implements Runnable {
   public void run() {
      synchronized(this) {
         try { wait();}
         catch (InterruptedException e) { System.out.println(e);}
      }
   }
   public static void main(String[] args) throws InterruptedException {
      Thread t = new Thread(new Simple());
      t.start();          System.out.println("Simple thread started.");
      t.interrupt();      System.out.println("Interrupt sent ...");
      t.join();           System.out.println("Joining simple thread.");
   }
}  // end class Simple
```

... liefert die Ausgabe:

```
Simple thread started.
Interrupt sent ...
java.lang.InterruptedException: operation interrupted
Joining simple thread.
```

- Für ein `Runnable` der Klasse `Simple` wird ein Thread gestartet.
- Der neue Thread reiht sich in die Warteschlange für das `Simple`-Objekt ein und geht in den Zustand waiting.
- Bei Ankunft des Interrupt wird der Thread aufgeweckt.
- Mit einem `lock()` betritt er seinen kritischen Abschnitt, um dort eine `InterruptedException` zu werfen.
- Der Thread beendet sich, und weckt sämtliche Threads seiner Schlange `joiningThreads`, d.h. den Thread `main`, auf.

Beachte:

- Ein laufender Thread kann Interrupts ignorieren :-)
- Ein wartender Thread kann die `InterruptedException` erst werfen, wenn er wieder das Lock erhielt :-)
- Wartet ein Thread (im Zustand joiningIO) auf Beendigung einer IO-Operation (z.B. auf Eingabe vom Terminal) und erhält ein Interrupt, dann wirft er eine `InterruptedIOException`.
- Keine Exception entkommt aus einem Thread.
- Die Objekt-Methoden `run()` der Klasse `Thread` wie des Interface `Runnable` deklarieren keinerlei Exceptions.
- `InterruptedExceptions` und `InterruptedIOExceptions` müssen darum wie alle anderen Exceptions innerhalb des Thread abgefangen werden !!!

Anwendung: Timeout für Terminal-Input

- Eingabe auf `System.in` blockiert die Programmausführung.
- Um Warten auf Eingabe zeitlich zu begrenzen, starten wir einen Thread der Klasse `TimeOut`.
- Dieser soll nach einer gewissen Zeit das Warten unterbrechen.
- Wurde die Eingabe-Operation allerdings vorher erfolgreich beendet, soll der `TimeOut`-Thread selbst beendet werden.
- Nützliche Methoden der Klasse `Thread`:
  - `public static boolean interrupted();`
  - `public boolean isInterrupted();`

  ... testen, ob das `interrupted`-Flag gesetzt ist, und setzen es dann auf false.

```
import java.io.*;
public class TimeOut extends Thread {
   private Thread t; private int msec;
   public TimeOut(int n) {
      t = Thread.currentThread(); msec = n;
   }
   public synchronized void run() {
      try {
         wait(msec);
         if(!Thread.interrupted()) t.interrupt();
      } catch (InterruptedException e) { }
   }
   public synchronized void kill() {
      if (!Thread.interrupted()) interrupt();
   }
   ...
```

- 1. Fall: Die Input-Operation wurde nicht unterbrochen. Dann wird die `synchronized`-Methode `kill()` aufgerufen.

  Stellt diese doch noch einen Interrupt fest, ist offenbar `TimeOut` beendet. Nichts muss passieren.

  Stellt diese keinen Interrupt fest, wartet der `TimeOut`-Thread (entweder in der Schlange `waitingThreads` oder hinter dem `wait()` auf die Fortsetzung des kritschen Abschnitts). Dann erhält er einen Interrupt. Kommt daraufhin der `TimeOut`-Thread in seinen kritischen Abschnitt, wird er sich einfach beenden.

- 2. Fall: Der `timeOut`-Thread beendet ungestört sein Warten und erwirbt erneut das Lock.

  Dann sendet er einen Interrupt und beendet sich. Die Operation `kill()` testet das Interrupt-Flag mithilfe der Klassen-Methode boolean Thread.interrupted() – und setzt es dadurch zurück.

```
    public static void echo(BufferedReader bu) {
       try { System.out.println(bu.readLine()); }
       catch (InterruptedIOException e) {
          System.err.println("Sorry, timeout!"); }
       catch (IOException e) {
          System.err.println("Sorry, general IO exception!"); }
    }
    public static void main(String[] args) {
       BufferedReader stdin = new BufferedReader
          (new InputStreamReader(System.in));
       TimeOut t = new TimeOut(5000);
       t.start();    System.out.println("TimeOut thread started.");
       echo(stdin);
       t.kill();     System.out.println("Timed input completed.");
    }
}  // end class TimeOut
```

... liefert z.B.:

```
> java TimeOut
TimeOut thread started.
abc
abc
Timed input completed.
```

... oder:

```
> java TimeOut
TimeOut thread started.
Sorry, timeout!
Timed input completed.
```

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.

21 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 :-)