Java-Threads kompetent einsetzen

Aegidius Plüss

www.aplu.ch

V1.2, Juni 2006

1 Synchronized

Schreibt man Programme mit einer grafischen Benützeroberfläche (GUI), wie es heute üblich ist, so ist man unmittelbar mit der Nebenläufigkeit (Multithreading) konfrontiert, denn bei Ereignissen auf der Benützeroberfläche (Button-Betätigungen, usw.) werden Methoden (Callbacks) von einem speziellen Java-Thread aufgerufen (dem Event Dispatch Thread, EDT). Fehlen die nötigen Kenntnisse über das Multithreading und hält man sich nicht an gewisse grundsätzliche Regeln, so können bereits einfache Programme ein fehlerhaftes oder zumindest ungewöhliches Verhalten aufweisen (zeitweises Einfrieren des GUI, usw.). Da es sich oft um Fehler handelt, die nur unter gewissen schlecht reproduzierbaren Umständen auftreten, sind sie grundsätzlich schwierig zu beheben. Wir betrachten im Folgenden eine exemplarische Lernsequenz, welche die Probleme aufdeckt und Lösungen anbietet. Wie so oft, setzen wir aus didaktischen Gründen zu Beginn die Turtle aus dem Package ch.aplu.turtle ein, denn ein Bild sagt bekanntlich mehr als tausend Worte.

1.1 Ausgangslage

Eine Turtle soll in einem Applikationsprogramm regelmässige Dreieckszacken zeichnen. Ein etwas faules "Stempelkind", das einen eigenen Thread und damit viel Eigenleben besitzt, wird zu bestimmten Zeiten das Turtlebild auf die Unterlage "stempeln". Nachher schläft es wieder ziemlich lange ein. Wie wir aus den Grundlagen der Threadprogrammierung entnehmen, leiten wir dazu die Klasse `StampKid` aus Thread ab und implementieren die run-Methode. Beim Start des Threads wird diese im neuen Thread laufen und soll, immer nach einer beträchtlichen Schlafpause, die Turtle stempeln. Bei Programmabbruch mit dem Close-Button wird vom Button-Callback die Methode `System.exit()` aufgerufen, wodurch auch der Thread des Stempelkindes beendet wird.

Damit das Stempelkind auf die Turtle zugreifen kann, um sie zu stempeln, übergeben wir ihm bei der Konstruktion eine Referenz der Applikationsklasseninstanz. Weil die Instanzvariable `Turtle t` keinen expliziten Zugriffsbezeichner besitzt ist, kann das Stempelkind sie benützen, da es sich im gleichen Package befindet.

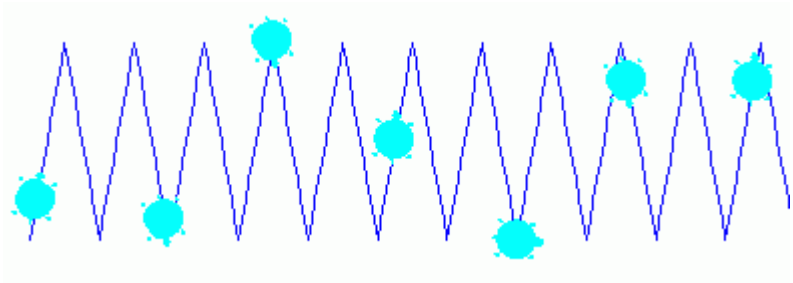
Im Applikationsprogramm wird die Turtle zuerst in eine günstige Anfangsposition gesetzt. Mit dem Aufruf von `start()` wird der Thread des Stempelkindes seine Arbeit aufnehmen, indem er im neu erzeugten Thread die run-Methoden ausführt. In einer Endlosschleife zeichnet die Turtle einen Dreieckszacken nach dem anderen.

```
// SynchEx1.java
// StampKid stamps whenever it likes

import ch.aplu.turtle.*;

public class SynchEx1
{
  Turtle t = new Turtle();

  public SynchEx1()
  {
    t.speed(100);
    t.setPos(-180, 0);
    t.right(10);
    new StampKid(this).start();
    while (true)
    {
      t.forward(100);
      t.right(160);
      t.forward(100);
      t.left(160);
    }
  }

  public static void main(String[] args)
  {
    new SynchEx1();
  }
}

// ---------------------- class StampKid ---------------------
class StampKid extends Thread
{
  SynchEx1 app;

  StampKid(SynchEx1 app)
  {
    this.app = app;
  }

  public void run()
  {
    while (true)
    {
      app.t.stampTurtle();
      try
      {
        Thread.currentThread().sleep(5000);
```

```
      }
      catch (InterruptedException ex) {}
    }
  }
}
```

1.1.1 Resultat

Das Resultat erstaunt wenig: das Stempelkind erwacht alle 5 Sekunden und stempelt die Turtle, ohne Rücksicht darauf, wo sich diese gerade befindet, sogar mitten in einer Drehung. Das Resultat ist entsprechend unkoordiniert, da der Applikationsthread und der Thread des Stempelkindes nicht synchronisiert sind.



1.1.2 Kommentar

Wir möchten das Stempelkind veranlassen, nur dann zu stempeln, wenn ein Dreieckszacken fertig gezeichnet ist. Dazu versuchen wir den Kindthread daran zu hindern, den Schleifenblock zu unterbrechen. Nach einer allgemeinen Lehrmeinung genügt es dazu, den betreffenden Block als `synchronized` zu bezeichnen. Wir fügen daher im ersten Versuch einzig einen synchronized-Block ein:

```
    while (true)
    {
      synchronized(this)
      {
        t.forward(100);
        t.right(160);
        t.forward(100);
        t.left(160);
      }
    }
```

1.1.3 Resultat

Das Resultat ist ernüchternd, wir stellen keinen Unterschied des Programmverhaltens fest.

1.1.4 Kommentar

Offenbar ist es ein Irrtum zu glauben, man könne einen Programmblock mit `synchronized` bezeichnen, um zu verhindern dass er von einem anderen Thread unterbrochen wird. Diese Vorstellung stammt möglicherweise aus den Zeiten der kooperativen Betriebsysteme, wo jedes Programm (bis auf Interrupts) den Code in alleiniger Herrschaft ausführen konnte und selbst entschied, ob es anderen Programmen die Ausführung gestattete. Es genügte damals, beim Eintritt in einen Codeteil, der nicht unterbrochen werden durfte, mit einem Befehl wie `noyield()` die Weitergabe abzuschalten. Diese Zeiten sind aber eindeutig passé. In modernen Multitasking-Systemen werden Prozesse und Threads vom Betriebssystem **preemptive** verwaltet, d.h. die Umschaltung erfolgt ohne viele Einflussmöglichkeiten durch das Anwenderprogramm (unter Windows werden Prozess-Prioritäten sehr schlecht unterstützt). Es lassen sich daher keine Programmblöcke mehr auszeichnen, die vor Unterbrechungen gänzlich geschützt sind. Man beschränkt sich vielmehr darauf, gemeinsam benützte **Daten** derart zu schützen, dass mehrere Threads **koordiniert** oder eben **synchronisiert** darauf zugreifen. Unter Daten versteht man hier Java-**Klasseninstanzen**, deren **Instanzvariablen** man durch geeignete Verfahren vor einem ungeordneten Zugriff (lesen oder verändern) durch verschiedene Threads schützen will.

Java stellt dazu das Schlüsselwort `synchronized` zur Verfügung. Mit ihm werden Programmblöcke oder ganze Methoden in mehreren verschiedenen Threads ausgezeichnet, die koordiniert auf ein und dasselbe Objekt zugreifen wollen. Die betreffende Objektreferenz wird `synchronized` als Parameter mitgegeben. Es handelt sich also **nicht** um einen **einseitigen Schutz** vor Unterbrechungen, sondern vielmehr um eine **gegenseitige Vereinbarung** über den geordneten Zugriff auf ein Objekt. Der Parameter kann auch `this` sein, wenn man die aktuelle Instanz und damit alle seine Instanzvariablen schützen will. Deklariert man eine ganze Methode als `synchronized`, entspricht dies einem `synchronized(this)` des ganzen Methodenrumpfs.

Um unsere Zielsetzung zu erreichen, müssen wir auch im Thread des Stempelkinds den Bereich, der auf die Applikationsinstanz zugreift, in einen synchronized-Block setzen.

```
public void run()
{
  while (true)
  {
    synchronized(this)
    {
      app.t.stampTurtle();
    }
    try
    {
```

```
        Thread.currentThread().sleep(5000);
      }
      catch (InterruptedException ex) {}
    }
  }
```

1.1.5 Resultat

Offensichtlich haben wir etwas Grundsätzliches falsch gemacht, denn das Resultat ist unverändert und das Stempelkind kann den Zackencode an beliebigen Stellen unterbrechen. Wo liegt der Fehler?

1.1.6 Kommentar

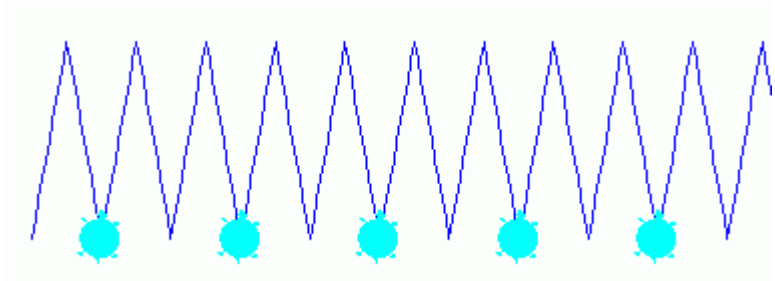
Wir haben in der Eile übersehen, dass die Objektreferenz, die wir `synchronized` mitgeben, das von beiden Threads zu schützende Objekt bezeichnet. Setzen wir in beiden Threads `synchronized(this)`, so werden dabei zwei verschiedene Klasseninstanzen referenziert, nämlich eine Instanz der Applikationsklasse und eine Instanz des Stempelkindes. Mit dieser Erkenntnis schreiben wir das korrekte Programm, indem wir auch im Stempelkind mit `synchronized(app)` die Applikationsklasse referenzieren.

```
// SynchEx2.java
// Finally it works!
// The synchronized block is now atomic (not interrupted)

import ch.aplu.turtle.*;

public class SynchEx2
{
  Turtle t = new Turtle();

  public SynchEx2()
  {
    t.speed(100);
    t.setPos(-180, 0);
    t.right(10);
    new StampKid(this).start();
    while (true)
    {
      synchronized(this)
      {
        t.forward(100);
        t.right(160);
        t.forward(100);
        t.left(160);
```

```
      }
    }
  }

  public static void main(String[] args)
  {
    new SynchEx2();
  }
}

// ---------------------- class StampKid ---------------------
class StampKid extends Thread
{
  SynchEx2 app;

  StampKid(SynchEx2 app)
  {
    this.app = app;
  }

  public void run()
  {
    while (true)
    {
      synchronized(app)
      {
        app.t.stampTurtle();
      }
      try
      {
        Thread.currentThread().sleep(5000);
      }
      catch (InterruptedException ex) {}
    }
  }
}
```

1.1.7 Resultat

Es wird jetzt nur noch an gewünschter Stelle gestempelt, leider allerdings nicht nach jedem Zacken.



1.1.8 Kommentar

Java implementiert den gegenseitigen Ausschluss mit `synchronized` wie folgt: Beim Eintritt in den mit `synchronized(obj)` bezeichneten Block (auch **kritischer Abschnitt kA** genannt) erhält der ausführende Thread vom Objekt `obj` eine **Sperre** (**Lock**, auch **Monitor** genannt). Den Monitor kann man sich auch als eine Art Schlüssel vorstellen, von dem es für jede **Objektinstanz** nur einen einzigen gibt. Die anderen Threads, welche etwas später ebenfalls in den *kA* eintreten wollen, müssen sich zuerst diesen Monitor beschaffen. Ist dieser bereits vergeben, so werden sie vom Threadscheduler in einen **blockierten** Zustand versetzt. Sobald der Monitor beim Verlassen des *kA* frei wird, gehen die blockierten Threads in den runnable-Zustand und versuchen, vom Threadscheduler den Monitor zu erhalten. Es ist nicht zum vornherein klar, welcher Thread diesen als ersten auch tatsächlich erhält und laufen wird. Es können aber alle blockierten Threads sicher sein, den Monitor zu kriegen, bevor der erste Thread wieder an die Reihe kommt, denn der Scheduler hält sich an das **Prinzip der Fairness**.

Das Verhalten mit mehreren Threads, die sich um den Monitor bemühen, können wir untersuchen, wenn wir zusätzlich zum Stempelkind noch ein Malkind deklarieren, das die Turtle rot oder grün färben kann. Dazu verwenden wir den Thread `PaintKid`, der in der run-Methode mit `setColor()` die Farbe der Turtle verändert.

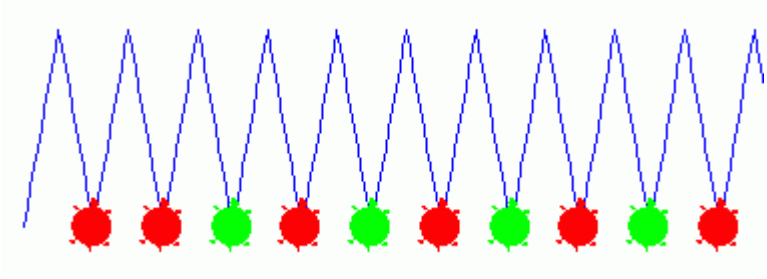
```
// SynchEx3.java
// If two threads are waiting, it's not sure who will be first

import ch.aplu.turtle.*;
import java.awt.Color;

public class SynchEx3
{
  Turtle t = new Turtle();

  public SynchEx3()
  {
    t.speed(100);
    t.setPos(-180, 0);
    t.right(10);
    t.setColor(Color.green);
```

```
    new PaintKid(this).start();
    new StampKid(this).start();
    while (true)
    {
      synchronized(t)
      {
        t.forward(100);
        t.right(160);
        t.forward(100);
        t.left(160);
      }
    }
  }

  public static void main(String[] args)
  {
    new SynchEx3();
  }
}

// ---------------------- class StampKid ---------------------
class StampKid extends Thread
{
  SynchEx3 app;

  StampKid(SynchEx3 app)
  {
    this.app = app;
  }

  public void run()
  {
    while (true)
    {
      synchronized(app.t)
      {
        app.t.stampTurtle();
      }
      try
      {
        Thread.currentThread().sleep(1000);
      }
      catch (InterruptedException ex) {}
    }
  }
}

// ---------------------- class PaintKid ---------------------
```

```
class PaintKid extends Thread
{
  SynchEx3 app;

  PaintKid(SynchEx3 app)
  {
    this.app = app;
  }

  public void run()
  {
    while (true)
    {
      synchronized(app.t)
      {
        if (app.t.getColor() == Color.red)
          app.t.setColor(Color.green);
        else
          app.t.setColor(Color.red);
      }
      try
      {
        Thread.currentThread().sleep(1300);
      }
      catch (InterruptedException ex) {}
    }
  }
}
```

1.1.9 Resultat

Die Turtle startet grün und wird nach der ersten und zweiten Kehrtwendung rot gestempelt. Offensichtlich hat bei der ersten Kehrtwendung das Malkind, bei der zweiten aber das Stempelkind Vorrang.



1.2 wait und notify

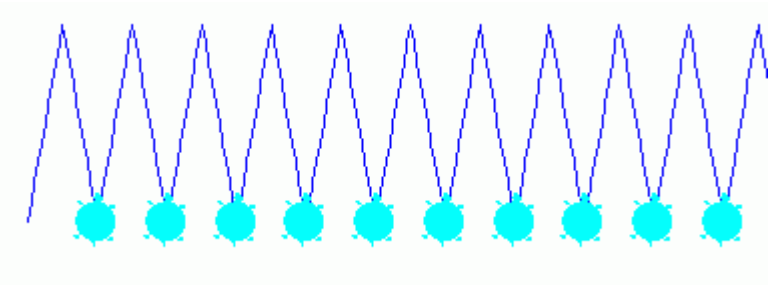
Zuletzt bleibt uns noch das Problem zu lösen, dafür zu sorgen, dass die Turtle nach jedem Zacken gestempelt wird. Offensichtlich muss der Applikationsthread auf das Aufwachen des Langschläfer-Stempelkinds warten, bevor er mit dem Zeichen des nächsten Zacken weiterfährt. Dazu stellt Java die Methoden `wait()` und `notify()` der Klasse `Object` zur Verfügung, die daher von allen Instanzen aufrufbar sind. Dabei müssen wir beachten, dass `wait()` nur von einem Objekt aufgerufen werden darf, das den Monitor vergeben hat, sich also in einem *kA* befindet, und dass `notify()` des gleichen Objekt aufgerufen werden muss, um den Thread weiter laufen zu lassen. Sobald `wait()` aufgerufen wird, geht der Thread in den Zustand **suspended** und gibt den Monitor vorübergehend ab. Mit `notify()` wird der Thread später genau an dieser Stelle weiter laufen.

```
// SynchEx4.java
// Wait for stampKid, to be sure, we are stamped

import ch.aplu.turtle.*;
import java.awt.Color;

public class SynchEx4
{
  Turtle t = new Turtle();

  public SynchEx4()
  {
    t.speed(100);
    t.setPos(-180, 0);
    t.right(10);
    new StampKid(this).start();
    while (true)
    {
      synchronized(this)
      { // We owe the monitor now
        t.forward(100);
        t.right(160);
        t.forward(100);
        t.left(160);
        try
        {
          wait();  // Suspend, until we get notified
        }
        catch (InterruptedException ex) {}
      }
    }
  }

  public static void main(String[] args)
```

```
  {
    new SynchEx4();
  }
}

// ---------------------- class StampKid ---------------------
class StampKid extends Thread
{
  SynchEx4 app;

  StampKid(SynchEx4 app)
  {
    this.app = app;
  }

  public void run()
  {
    while (true)
    {
      synchronized(app)
      {
        app.t.stampTurtle();
        app.notify();  // Notify the suspended app
      }
      try
      {
        Thread.currentThread().sleep(5000);
      }
      catch (InterruptedException ex) {}
    }
  }
}
```

1.2.1 Resultat

Wir haben endlich erreicht, was wir ursprünglich beabsichtigten und die Turtle wird nach jedem Zacken gezeichnet.



1.2.2 Kommentar

Das Zeichnen des Zackens erfolgt in einem Block, der durch den Stempelthread nicht unterbrochen werden kann. Dasselbe würden wir erreichen, falls wir eine synchronisierte Methode `zacken()` deklarieren. Die Änderung betrifft folgenden Programmteil:

```
  public SynchEx4()
  {
    t.speed(100);
    t.setPos(-180, 0);
    t.right(10);
    new StampKid(this).start();
    while (true)
      zacken();
  }

  synchronized void zacken()
  {
    t.forward(100);
    t.right(160);
    t.forward(100);
    t.left(160);
    try
    {
      wait();
    }
    catch (InterruptedException ex) {}
  }
```

Wir haben nun allerdings des Guten zu viel getan, denn wir müssen eigentlich gar nicht die ganze Applikationsinstanz, sondern lediglich die Turtleinstanz vor dem gemeinsamen Zugriff schützen. Es ist aber leicht, das Programm entsprechend zu modifizieren. Das Resultat bleibt das gleiche.

```
// SynchEx5.java
// Protect Turtle only

import ch.aplu.turtle.*;

public class SynchEx5
{
  Turtle t = new Turtle();

  public SynchEx5()
  {
    t.speed(100);
    t.setPos(-180, 0);
    t.right(10);
```

```
    new StampKid(t).start();
    while (true)
    {
      synchronized(t)
      {
        t.forward(100);
        t.right(160);
        t.forward(100);
        t.left(160);
        try
        {
          t.wait();
        }
        catch (InterruptedException ex) {}
      }
    }
  }

  public static void main(String[] args)
  {
    new SynchEx5();
  }
}

// ---------------------- class StampKid ---------------------
class StampKid extends Thread
{
  Turtle t;

  StampKid(Turtle t)
  {
    this.t = t;
  }

  public void run()
  {
    while (true)
    {
      synchronized(t)
      {
        t.stampTurtle();
        t.notify();
      }
      try
      {
        Thread.currentThread().sleep(5000);
      }
      catch (InterruptedException ex) {}
```

```
    }
  }
}
```

2 Atomar, volatile und thread-safe

2.1 64-bit-Basistypen

Man bezeichnet einen Ausdruck oder einen Codeabschnitt als **atomar**, falls er nicht durch Threads unterbrochen werden kann, die einen Einfluss auf das Resultat haben können. In Java sind bereits gewöhnliche Operationen auf 64-bit lange Basistypen, also auf long und doubles nicht atomar. Mit anderen Worten: Greifen zwei Threads auf nicht atomare Instanzvariablen zu, so muss damit gerechnet werden, dass es **sehr selten völlig falsche Resultate** gibt, da der Thread-Wechsel dann erfolgen kann, wenn erst der eine der beiden (atomaren) 32-bit-Teile verarbeitet ist. Wir zeigen dieses Fehlverhalten in einem Programm, das laufend in zwei verschiedenen Threads den oberen und unteren 32-bit-Teil eines long kehrt (swap), wobei im einen Teil alle Bits gesetzt und im anderen alle Bits gelöscht sind. Erfolgt der Threadwechsel, wenn erst der eine Teil gekehrt ist, so ergibt sich eine Zahl, bei der alle 64 Bits gesetzt oder gelöscht sind, was den long-Zahlen -1 bzw. 0 entspricht.

```
// AtomicEx1.java
// Operation may fail at different moments
// depending on CPU speed and operating system
// On failure we may get i = 0 (0x00000000'00000000)
// or i = -1 (0xFFFFFFFF'FFFFFFFF)

public class AtomicEx1
{
  long i = 0x00000000FFFFFFFFL;

  public AtomicEx1()
  {
    long n = 0;
    new UpdateThread(this).start();
    while (true)
```

```
    {
      if (n % 10000000L == 0)  // Adapt to speed of your PC!!
        System.out.println(n);
      n++;
      i = ~i;  // 1's complement: swap low and high 32 bits

      // Inhibit interruption while testing
      synchronized(this)        {
        if (!(i == 4294967295L || i == -4294967296L))
        {
          System.out.println("Swap failed. i = " + i);
          System.exit(0);
        }
      }
    }
  }

  public static void main(String[] args)
  {
    new AtomicEx1();
  }
}

// ----------------- class UpdateThread ---------------------
class UpdateThread extends Thread
{
  AtomicEx1 app;

  UpdateThread(AtomicEx1 app)
  {
    this.app = app;
  }

  public void run()
  {
    while (true)
    {
      synchronized(app)
      {
        app.i = ~app.i;
      }
    }
  }
}
```

2.1.1 Resultat

Man muss das Programm meist mehrmals starten und lange laufen lassen, bis der Fehler auftritt.

2.1.2 Kommentar

Solche Fehler sind besonders heimtückisch, da sie selten auftreten und selbst in einer umfangreichen Testphase nicht in Erscheinung treten, sich aber im operativen Einsatz des Programms katastrophal auswirken können. Es genügt, die Instanzvariable `volatile` zu deklarieren, um den Fehler zu vermeiden. Das Schlüsselwort `volatile` hat also zwei Bedeutungen: Zum einen bewirkt es, dass die Operationen mit einem `long` oder `double` immer atomar bleiben, zum anderen wird der Compiler aufgefordert, die so bezeichneten Variablen in keinem Fall zu optimieren, d.h. durch einen konstanten Wert zu ersetzen, selbst wenn sie vom aktuellen Thread nicht verändert werden. Instanzvariablen, die von mehreren Threads verwendet werden, sollten daher **immer volatile** deklariert werden. Setzen wir

```
volatile long i = 0x00000000FFFFFFFFL;
```

so ist tritt der Fehler nie mehr auf. Auf Grund unserer Vorkenntnisse wissen wir, dass der Fehler auch behoben werden kann, indem wir die Swap-Operation in beiden Threads in einen `synchronized`-Block zu setzen.

```
// AtomicEx3.java
// Same as AtomicEx1, but synchronize swap operation
// Operation never fails but slow execution speed

public class AtomicEx3
{
  long i = 0x00000000FFFFFFFFL;

  public AtomicEx3()
  {
    long n = 0;
    new UpdateThread(this).start();
    while (true)
    {
      if (n % 10000000L == 0)
        System.out.println(n);
      n++;
      synchronized(this)
      {
        i = ~i;
        if (!(i == 4294967295L || i == -4294967296L))
          throw new RuntimeException("Swap failed. i = " + i);
      }
    }
```

```
  }

  public static void main(String[] args)
  {
    new AtomicEx3();
  }
}

// ---------------------- class UpdateThread ----------------
class UpdateThread extends Thread
{
  AtomicEx3 app;

  UpdateThread(AtomicEx3 app)
  {
    this.app = app;
  }

  public void run()
  {
    while (true)
    {
      synchronized(app)
      {
        app.i = ~app.i;
      }
    }
  }
}
```

2.2 Atomisieren einer Klasse

Es liegt nahe, die Schwierigkeiten beim gemeinsamen Zugriff mehrerer Threads zu vermeiden, indem man grundsätzlich alle Methoden einer Klasse `synchronized` deklariert. Beispielsweise kann eine `AtomicTurtle` erstellt werden, die man aus der Klasse `Turtle` ableitet und in der man alle public Methoden durch gleichlautende Methoden, die `synchronized` deklariert sind, überschreibt. Wir führen dies an vier Methoden durch.

```
// AtomicTurtle.java

import ch.aplu.turtle.*;

public class AtomicTurtle extends Turtle
```

```
{
  synchronized public AtomicTurtle forward(double d)
  {
    super.forward(d);
    return this;
  }

  synchronized public AtomicTurtle left(double a)
  {
    super.left(a);
    return this;
  }

  synchronized public AtomicTurtle right(double a)
  {
    super.right(a);
    return this;
  }

  synchronized public AtomicTurtle stampTurtle()
  {
    super.stampTurtle();
    return this;
  }
}
```

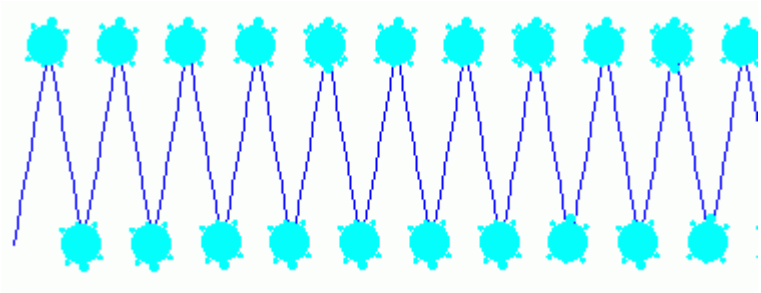
Schreiben wir eine Applikation, die statt einer `Turtle` eine `AtomicTurtle` die Zacken zeichnen lässt, so kann auch ein fleißiges Stempelkind (das kein `sleep()` enthält) nur zwischen den einzelnen Methodenaufrufen stempeln, dort vielleicht mehrmals.

```
// AtomicEx4.java

public class AtomicEx4
{
  AtomicTurtle at = new AtomicTurtle();

  public AtomicEx4()
  {
    at.speed(100);
    at.setPos(-180, 0);
    at.right(10);
    new RapidStampKid(this).start();
    while (true)
    {
      at.forward(100);
      at.right(160);
      at.forward(100);
      at.left(160);
```

```
    }
  }

  public static void main(String[] args)
  {
    new AtomicEx4();
  }
}

// ----------------- class RapidStampKid ---------------------
class RapidStampKid extends Thread
{
  AtomicEx4 app;

  RapidStampKid(AtomicEx4 app)
  {
    this.app = app;
  }

  public void run()
  {
    while (true)
    {
      app.at.stampTurtle();
    }
  }
}
```

2.2.1 Resultat

Man erkennt, dass das Stempelkind zwar **nicht während** der Ausführung von `forward()`, `left()` oder `right()` stempeln kann, hingegen sehr wohl **zwischen** den einzelnen Aufrufen.



2.2.2 Kommentar

Dieses summarische Verfahren, alle Methoden, unabhängig davon, ob dies nötig ist oder nicht, mit `synchronized` zu deklarieren, ist sicher wenig elegant. Da die Laufzeit dadurch stark zunehmen kann, bestraft man damit Methoden, die es gar nicht verdienen, mit einer Performanzeinbusse. Fast noch schlimmer ist die Tatsache, dass die vielen mit synchronized deklarierten Methoden zu völlig unerwarteten und gefürchteten **Deadlocks** führen können, bei denen mehrere Threads **gegenseitig aufeinander warten**.

2.3 Das Package java.util.concurrent.atomic

Ein klassisches Beispiel, welches die Notwendigkeit von atomaren Ausdrücken aufzeigt, ergibt sich, wenn wir eine Klasse `Counter` deklarieren, die als Instanzvariable einen aktuellen Zählerstand aufweist, den man mit drei Methoden abfragen, inkrementieren und dekrementieren kann. Wir betrachten zuerst die folgende Implementierung, die selbsterklärend ist:

```
// ---------------------- class Counter --------------------
class Counter
{
  private int value = 0;

  public int increment()
  {
    return ++value;
  }

  public int decrement()
  {
    return --value;
  }

  int getValue()
  {
    return value;
  }
}
```

Den Counter verwenden wir in eine Applikation, die sehr praxisnahe ist. Sie simuliert eine Bank mit einem Bankkonto, auf das zwei zahlungskräftige Kunden unermüdlich Geld in Form von gleichen Geldstücken einzahlen. Das Konto wird durch eine einzige Counterinstanz implementiert, die den Kunden zum Inkrementieren zur Verfügung gestellt wird. Die Kunden führen aber natürlich auch eine eigene Buchhaltung, wissen also, wie oft sie ein Geldstück überwiesen haben. Nach einer gewissen Zeit schließt das Bankinstitut und

vergleicht in einem Ausdruck den Stand des Bankkontos mit den beiden Kundenbuchhaltungen.

```
// AtomicEx5.java
// Counter fails, because two threads access
// non-atomic ++value
// (Counter value too small)

public class AtomicEx5
{
  private Counter counter;

  public AtomicEx5()
  {
    counter = new Counter();
    User user1 = new User(counter);
    User user2 = new User(counter);
    user1.start();  // Start users
    user2.start();
    try
    {
      Thread.currentThread().sleep(3000); // Let them work
      user1.halt(); // Stop them
      user2.halt();
      user1.join(); // and wait until they are dead
      user2.join();
    }
    catch (InterruptedException ex) {}

    int value1 = user1.getValue();
    int value2 = user2.getValue();
    int total = value1 + value2;

    System.out.println("user1 #:   " + value1);
    System.out.println("user2 #:   " + value2);
    System.out.println("total #:   " + total);
    System.out.println("counter #: " + counter.getValue());
    System.out.println("Max: " + Integer.MAX_VALUE);
  }

  public static void main(String[] args)
  {
    new AtomicEx5();
  }
}

// ---------------------- class User -------------------------
-
```

```
class User extends Thread
{
  private Counter counter;
  private int value = 0;
  private boolean isRunning = false;

  User(Counter counter)
  {
    this.counter = counter;
  }

  int getValue()
  {
    return value;
  }

  void halt()
  {
    isRunning = false;
  }

  public void run()
  {
    isRunning = true;
    while (isRunning)
    {
        counter.increment();
        value++;
    }
  }
}
```

2.3.1 Resultat

Das Resultat ist erschreckend: Die Bilanz stimmt überhaupt nicht, auf dem Bankkonto liegt zuwenig Geld. Typisch erhalten wir

```
user1 #:   424977510
user2 #:   420733969
total #:   845711479
counter #: 648380922
Max: 2147483647
```

2.3.2 Kommentar

Obschon die beiden Threads nur die einfache Methode `increment()` aufrufen, ergeben sich massive Schwierigkeiten. Offenbar ist der Ausdruck `++value` nicht atomar; in unglücklichen Momenten kann es vorkommen, dass die beiden Threads vor dem Inkrementieren den gleichen aktuellen Wert erhalten und diesen gleichen Wert erhöhen. Nach dem Zurückspeichern ist der Wert dann nur einmal erhöht, was den zu kleinen Counterwert erklärt.

Als einfachste Gegenmaßnahme atomisieren wir den Counter, indem wir alle Methoden mit `synchronized` versehen:

```
// ------------------- class Counter ------------------------
class Counter extends Thread
{
  private int value = 0;

  synchronized public int increment()
  {
    return ++value;
  }

  synchronized public int decrement()
  {
    return --value;
  }

  synchronized int getValue()
  {
    return value;
  }
}
```

2.3.3 Resultat

Wie erwartet, stimmt nun die Bilanz. Ein typischer Ausdruck ist:

```
user1 #:   15095686
user2 #:   9792542
total #:   24888228
counter #: 24888228
Max: 2147483647
```

2.3.4 Kommentar

Das Synchronisieren bewirkt, dass beim Aufruf von `increment()` der Monitor (die Sperre, Lock) der Counter-Instanz frei sein muss, ansonsten der betreffende Thread blockiert wird. Dieser muss warten, bis die Sperre aufgehoben wird, was dann erfolgt, wenn der andere Thread `increment()` beendet. Sogar `getValue()` sollte `synchronized` deklariert werden, denn `value` ist erst dann in einem gesicherten Zustand, wenn der Monitor wieder frei wird.

Wie wir vorhin feststellten, ist das Atomisieren durch Synchronisieren keine Patentlösung. Aus diesem Grund wird in den neueren Java-Versionen ein Package zur Verfügung gestellt, das Klassen mit atomaren Versionen der Basistypen enthält. Einige grundlegende Operationen werden mit direkter Unterstützung durch die Rechner-Plattform atomar ausgeführt und sind daher wesentlich effizienter. Die korrekte Verwendung der Klassen ist keineswegs trivial. Immerhin lässt sich damit der Counter ohne Verwendung von `synchronized` realisieren, wodurch auch Deadlocks vermieden werden.

```
// AtomicEx7.java
// Uses new atomic package from J2SE V1.5 up
// No locks and waits needed
// Final values show significiant increase in speed
// (mean from several runs)

import java.util.concurrent.atomic.AtomicInteger;

public class AtomicEx7
{
  private Counter counter;

  public AtomicEx7()
  {
    counter = new Counter();
    User user1 = new User(counter);
    User user2 = new User(counter);
    user1.start();
    user2.start();
    try
    {
      Thread.currentThread().sleep(3000);
      user1.halt();
      user2.halt();
      user1.join();
      user2.join();
    }
    catch (InterruptedException ex) {}

    int value1 = user1.getValue();
    int value2 = user2.getValue();
```

```
    int total = value1 + value2;

    System.out.println("user1 #:   " + value1);
    System.out.println("user2 #:   " + value2);
    System.out.println("total #:   " + total);
    System.out.println("counter #: " + counter.getValue());
    System.out.println("Max: " + Integer.MAX_VALUE);
  }

  public static void main(String[] args)
  {
    new AtomicEx7();
  }
}

// --------------------- class User -------------------------
class User extends Thread
{
  private Counter counter;
  private int value = 0;
  private boolean isRunning = false;

  User(Counter counter)
  {
    this.counter = counter;
  }

  int getValue()
  {
    return value;
  }

  void halt()
  {
    isRunning = false;
  }

  public void run()
  {
    isRunning = true;
    while (isRunning)
    {
        counter.increment();
        value++;
    }
  }
}
```

```
// -------------------- class Counter -----------------------
class Counter extends Thread
{
  private AtomicInteger value = new AtomicInteger();

  public int increment()
  {
    int oldValue = value.get();
    while (!value.compareAndSet(oldValue, oldValue + 1))
      oldValue = value.get();
    return oldValue + 1;
  }

  public int decrement()
  {
    int oldValue = value.get();
    while (!value.compareAndSet(oldValue, oldValue - 1))
      oldValue = value.get();
    return oldValue - 1;
  }

  int getValue()
  {
    return value.get();
  }
}
```

2.3.5 Kommentar

Im den Code zu verstehen, muss man die JavaDoc des Package `java.util.concurrent.atomic` konsultieren; insbesondere ist die Verwendung von `compareAndSet()` ziemlich trickreich.

2.4 Sichere Threads (thread-safe)

Ein Codestück, eine Methode, eine Klasse oder eine ganze Klassenbibliothek wird dann als **thread-safe** bezeichnet, wenn mehrere Threads den gleichen Code ohne Rücksicht aufeinander verwenden können. Synonym dazu verwendet man **reentrant**, womit man ausdrückt, dass ein zweiter Thread mit der Ausführung eines Codeteil beginnen ("in den Code eintreten") kann, auch wenn ein erster denselben Codeteil noch nicht vollständig ausgeführt hat. Code, der thread-safe geschrieben ist, kann also sozusagen von mehreren Threads gleichzeitig ausgeführt werden, ohne dass sich diese in irgendeiner Art gegenseitig beeinflussen oder sich synchronisieren müssen. (Echte Gleichzeitigkeit gibt es nur in

Multiprozessor-Systemen.) Grundsätzlich gilt, dass die rein-funktionale Programmierung (ohne Seiteneffekte) immer zu thread-sicherem Code führt, und zwar deshalb, weil dabei lediglich lokale Variablen gebraucht werden. Jeder Methodenaufruf erhält seinen eigenen Speicherbereich für lokale Variablen (**Stack**), sodass sich lokale Variablen nicht gegenseitig beeinflussen können. Im Gegensatz dazu gelten alle globalen Variablen, in Java also die Instanzvariablen für die Threadsicherheit als gefährlich und sollten schon deswegen und nicht nur wegen des schlechten Programmierstils, wo immer möglich vermieden werden.

Bereits im vorhergehenden Beispiel haben wir aus einer nicht thread-sicheren Klasse Counter eine thread-sichere gemacht, indem wir alle Methoden als `synchronized` deklarierten und damit die gegenseitige Beeinflussung der Instanzvariable `value` durch mehrere Threads ausschlossen.

Wir simulieren einen Ablauf, den man in einer Schulklasse spielen lassen könnte. Einem Schüler wird von zwei verschiedenen Lehrern die Aufgabe gestellt, die n ersten natürlichen Zahlen zu addieren, also bei Vorgabe von n die Summe 1 + 2 + ... + n zu berechnen. Der Schüler soll dabei die einzelnen Zahlen zu einer Gesamtsumme addieren und diese jeweils auf einer Tafel aufschreiben. Da er nur wenig Platz hat, löscht er jeweils den vorhergehenden Wert. Da die Tafel in Java einer Instanzvariablen entspricht, sieht die Klasse `Pupil` so aus:

```
class Pupil
{
  int sum;

  int calculate(int n)
  {
    sum = 0;
    for (int i = 1; i <= n; i++)
      sum = sum + i;
    return sum;
  }
}
```

Zwei verschiedene Lehrer fragen den Schüler unermüdlich aus, indem sie für sich eine Zufallszahl zwischen 1 und 100 generieren und sie dem Schüler zur Bearbeitung übergeben. Der Schüler liefert das Resultat dem Lehrer ab und dieser überprüft es, natürlich nicht mit dem wenig geistreichen Verfahren des Schülers, sondern vielmehr mit der bekannten Formel n*(n+1)/2, die gemäss einer Anekdote der geniale Mathematiker C.F. Gauß bereits in der Grundschule herausgefunden hat. Gibt der Schüler ein falsches Resultat zurück, so wird der Vorgang abgebrochen.

```
// AtomicEx8.java
// Pupil is NOT thread-safe

import java.util.Random;

public class AtomicEx8
{
```

```
  private Pupil pupil = new Pupil();

  public AtomicEx8()
  {
    new Teacher("John", pupil).start();
    new Teacher("Emily", pupil).start();
  }

  public static void main(String[] args)
  {
    new AtomicEx8();
  }
}

class Teacher extends Thread
{
  private String name;
  private Pupil pupil;

  Teacher(String name, Pupil pupil)
  {
    this.name = name;
    this.pupil = pupil;
  }

  public void run()
  {
    int nb = 0;
    Random rand = new Random();
    while (true)
    {
      nb++;
      int n = 1 + rand.nextInt(100);
      int sum = pupil.calculate(n);
      if (sum != n * (n+1) / 2)
      {
         System.out.println(name + " says: failed after "
                                 + nb + " questions");
         System.exit(0);
      }
    }
  }
}
```

2.4.1 Resultat

Ein typischer Durchlauf ergibt folgenden Ausdruck

```
John says: failed after 517890 questions
Emily says: failed after 496892 questions
```

Es ist auch möglich, dass nur der eine Lehrer den Misserfolg meldet:

```
Emily says: failed after 1818149 questions
```

2.4.2 Kommentar

Wie vermutet, ist die Klasse `Pupil` nicht thread-safe, weil die Instanzvariable `sum` ein Zwischenresultat enthält, das bei einem Thread-Wechsel zerstört wird. Deklariert man, wie es sich gehört, die Variable `sum` als lokale Variable in der Methode `calculate()`, so ist das Problem gelöst. Übrigens könnte der Schüler die Summe auch rekursiv berechnen.

3 Swing und Threads

3.1 Swing-Regeln

Swing stellt ein riesiges Arsenal von Hilfsmitteln zur Erstellung von grafischen Benützeroberflächen (GUI) zur Verfügung. Konzept und Realisation sind aus mehreren Gründen nicht unumstritten, es gibt aber wenige echte Alternativen. Aus diesem Grund gehört Swing heute zu den Grundlagen von Java. Eine zentrale Rolle bei der Entwicklung von Swing-Applikationen spielen einerseits die Darstellung und Auffrischung der Grafik-Komponenten, andererseits die Programmlogik zur Notifikation von Benützeraktionen, welche im Wesentlichen von Mausereignissen ausgeben. Ersteres umfasst die vielfältigen Möglichkeiten des Layouts von Komponenten in einem Bildschirmfenster (Window) bzw. einem Dialog und das Repainting, letzteres die Behandlung von Callback-Methoden im Event-Listener-Modell von Java.

In jedem Java-Programm laufen mindesten drei Threads: der **Haupt-Thread** (main-thread), welcher in der main-Methode startet, der **Event Dispatch Thread** (**EDT**), welcher für die Behandlung der Fenster zuständig ist, und der **Garbage Collecting Thread**, welcher den

Speicherbereich aufräumt. Werden bei Programmen mit einem GUI die folgenden zwei grundlegende Regeln nicht konsequent eingehalten, so können thread-bedingte Fehler auftreten, die ihrer sporadischen Natur wegen oft schlecht reproduzierbar sind und eine sonst perfekte Applikation unbrauchbar machen.

Regel 1: "Laufzeit-Regel" (execution-time rule)

Listener-Methoden müssen in kurzer Zeit (maximal in einigen Hundert Millisekunden) zu Ende laufen, weil während des Ablaufs das GUI blockiert ist.

Regel 2: "Einzelthread-Regel" (single-thread rule)

Sobald ein GUI sichtbar ist, müssen alle Aufrufe von Swing-Methoden in einem einzigen Thread, nämlich dem EDT, erfolgen, weil Swing nicht thread-safe ist.

Von der zweiten Regel gibt es wenige Ausnahmen: sowohl `repaint()` und `revalidate()`, sowie alle Registrierungsmethoden `add...Listener()` brauchen nicht im EDT aufgerufen zu werden. Obschon es einige Swing-Methoden gibt, die thread-safe sind (beispielsweise `setText()`), sollten auch diese nicht außerhalb des EDT aufgerufen werden.

3.2 Swing-Programmierung

Wie wir sehen werden, komplizieren die beiden Regeln das Erstellen von GUI-basierte Programmen erheblich, denn sie sind in gewisser Hinsicht widersprüchlich: Die Laufzeit-Regel fordert kurze Listener-Methoden, die Einzelthread-Regel verlangt, dass dennoch alle GUI-Aktionen im Thread dieser Methoden ausgeführt werden. Es ergeben sich daraus für die Praxis folgende Richtlinien für Events, die von Swing-Komponenten ausgelöst werden:

- Kurz dauernde Aktionen (bis wenige Hundert Millisekunden) sind ausschließlich in den Listener-Methoden auszuführen. Da das GUI während dieser Zeit blockiert, verhält sich dieses wie eine modaler Dialog
- Länger dauernde Aktionen sind in einem besonderen User-Thread auszuführen, der asynchron zum EDT läuft. Da das GUI während dieser Ausführung wie ein nicht modaler Dialog wieder bedienbar wird, muss mit besonderen Vorsichtsmassnahmen (Inaktivieren von GUI-Komponenten, usw.) darauf geachtet werden, dass keine Mehrfach-Aktivitäten möglich sind, die zu einem falschen Programmverhalten führen. Im User-Thread dürfen für Ablauf- und Rückmeldungen keine direkten Aufrufe von Swing-Methoden gemacht werden. Diese müssen vielmehr dem EDT mit den Methoden `SwingUtilities.-invokeLater()` oder `SwingUtilities.invokeAndWait()` zur Ausführung übergeben werden. An Stelle eines User-Threads kann auch der Haupt-Thread (Applikationsthread) verwendet werden.

3.3 Ein Musterbeispiel

Im Folgenden werden die beiden Richtlinien an Hand von lauffähigen Musterbeispielen illustriert. Da es hier nur um Grundprinzipen geht, wird durchgängig ein sehr einfaches GUI erzeugt, das aus einem kurzen einzeiligen Textfeld (`JTextField`) für die Ein- und Ausgabe, einem Button *Go* und einem Button *Close* besteht. Mit dem Go-Button soll ein Vorgang ausgelöst werden, der zu Demonstrationszwecken seine Aktivität in einem Console-Fenster anzeigt und mit einer Zeitverzögerung einen länger dauernden Ablauf simuliert. Mit dem Close-Button wird das Programm beendet. Als Erweiterung soll der Go-Button nach dem Starten zu einem Stop-Button werden, mit dem der länger laufende Vorgang abgebrochen werden kann. Dieses Pflichtenheft ist insbesondere für Echtzeit- und Simulationsprogramme typisch, bei denen dem Benützer über eine längere Zeit Zwischenresultate, oft auch in Form von Fortschrittsbalken (progress bar) rückgemeldet werden.

Die Erzeugung des Dialog-Fensters mit Swing ist klassisch: In die `ContentPane` eines `JFrame` wird ein `JPanel` eingebettet, das ein `JLabel`, ein `JTextField` und zwei `JButton` enthält. Die entsprechenden Controls sind Instanzvariablen, weil meist auch in den Callback-Methoden des Listeners darauf zugegriffen werden muss. Die Initialisierung erfolgt in den Methoden `createGui()` und `init()`. Das Layout ist sehr einfach:

Für die Notifikation der Button-Events implementieren wir in der inneren Klasse `ButtonActionAdapter` das `ActionListener`-Interface und deklarieren deshalb die Listener-Methode `actionPerformed()`. Mit `addActionListener()` wird den Buttons eine Referenz dieser Klasse übergeben, damit sie beim Drücken als Callback die Listener-Methode aufrufen. Innerhalb der Listener-Methode muss noch untersucht werden, von welchem Button der Event stammt. Nachfolgend wird die entsprechende Callback-Methode `goButton_actionPerformed()` bzw. `closeButton_actionPerformed()` aufgerufen. Die Simulationsaktion besteht der Einfachheit halber aus einem Nickerchen von einer halben Sekunde. Zu Testzwecken wird der Simulationsschritt noch auf der Console ausgeschrieben.

Das erste Programm erstellen wir ohne Berücksichtigung der Richtlinien und führen die Simulation in der Callback-Methode des Go-Buttons durch.

```
// SwingEx1.java
// Conforms with single-thread rule,
// but not with execution-time rule

import java.awt.event.*;
import javax.swing.*;
```

```
public class SwingEx1
{

// -------------- class ButtonActionAdapter -----------------
private class ButtonActionAdapter implements ActionListener
{
  public void actionPerformed(ActionEvent evt)
  {
    if (evt.getSource() == goButton)
      goButton_actionPerformed();
    if (evt.getSource() == closeButton)
      closeButton_actionPerformed();
  }
}

// -------------- Application class ------------------------
  private JFrame frame;
  private JPanel controlPane = new JPanel();
  private JLabel labelSteps =
    new JLabel("Number of Steps (1..20): ");
  private JTextField textFieldSteps = new JTextField("10", 3);
  private JButton goButton = new JButton("Go");
  private JButton closeButton = new JButton("Close");

  public SwingEx1()
  {
    createGui();
  }

  private void createGui()
  {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                          WindowConstants.EXIT_ON_CLOSE);
    init();
    frame.pack();
    frame.setVisible(true);
  }

  private void init()
  {
    controlPane.add(labelSteps);
    controlPane.add(textFieldSteps);
    controlPane.add(goButton);
    controlPane.add(closeButton);
    frame.getContentPane().add(controlPane);
    goButton.addActionListener(new ButtonActionAdapter());
    closeButton.addActionListener(new ButtonActionAdapter());
```

```
  }

  private void goButton_actionPerformed()
  {
    int nbSteps;
    String steps = textFieldSteps.getText().trim();
    nbSteps = Integer.parseInt(steps);
    simulate(nbSteps);
  }

  private void closeButton_actionPerformed()
  {
    System.exit(0);
  }

  private void simulate(int nbSteps)
  {
    for (int step = 0; step < nbSteps; step++)
    {
      System.out.println("Working hard step # " + (step+1));
      delay(500); // Do something more intelligent...
    }
    textFieldSteps.setText("done");
  }

  private void delay(int ms)
  {
    try
    {
      Thread.currentThread().sleep(ms);
    }
    catch (InterruptedException ex) {}
  }

  public static void main(String[] args)
  {
    new SwingEx1();
  }
}
```

3.3.1 Resultat

Wie zu erwarten war, bleibt der Go-Button während der Simulation in der gedrückten Stellung und außer den Maximize/Minimize-Buttons ist während dieser Zeit das GUI nicht bedienbar und es können keine Zwischenwerte, beispielsweise mit `setText()`, in das GUI zurückgeschrieben werden. Wenig benützerfreundlich ist auch der Programmabsturz,

wenn der Anwender keinen Integer im Eingabefeld eingibt. Elegant lässt sich dieser vermeiden, wenn wir die Exception abfangen, die bei der Umwandlung des Strings in einen Integer geworfenen wird.

3.3.2 Kommentar

Im nächsten Verbesserungsschritt nehmen wir die zweite Richtlinie ernst und verlagern die Simulationsarbeit in den Haupt-Thread. Wie bei event-gesteuerten Programmen üblich, fassen wir verschiedenen Phasen des Programms mit den dazu gehörenden unterschiedlichen Erscheinungsbildern des GUI als **Zustände** im Sinn der Automatentheorie auf. Zu diesem Zweck führen wir eine **Zustandsvariable** ein, deren Wert durch die Events verändert wird. Ein Aufzählungstyp eignet sich dazu besonders gut, weil man ihm verbale Werte geben kann. Da die Zustandsvariablen in zwei verschienenen Threads gebraucht wird, sollte sie `volatile` deklariert sein. Ab Java Version 5 wird ein Aufzählungstyp wie folgt deklariert:

```
enum State {IDLE, RUNNING}
```

In früheren Java-Versionen, bei denen der Aufzählungstyp noch nicht zur Syntax gehört, kann ein Interface verwendet werden, um die Ganzzahl-Werte verbal auszudrücken:

```
interface State
{
  int IDLE = 0;
  int RUNNING = 1;
}
```

Im Haupt-Thread lassen wir eine **Ereignisschleife** (**event loop**) laufen, welche zur Steigerung der Effizienz im Zustand IDLE mit `Thread.sleep()` eine kurze Zeit den aktuellen Thread abgibt. Um der Einzelthread-Regel zu entsprechen, dürfen wir in der Ereignisschleife, die ja nicht im EDT läuft, keine Swing-Methoden aufrufen, sondern müssen folgenden Umweg einschlagen:

Wir setzten den Swing-Aufruf `setText()` in die run-Methode einer Klasse `SetTextControl`, welche `Runnable` implementiert. An Stelle des direkten Aufrufs übergeben wird der statischen Methode `SwingUtilities.invokeLater()` eine neu erstellte Instanz von `SetTextControl()`, was den EDT beauftragt, so bald als möglich die run-Methode auszuführen.

```
// SwingEx2.java
// Simulation in main-thread, compiles V1.5 up

import java.awt.event.*;
import javax.swing.*;

public class SwingEx2
{
```

```
enum State {IDLE, RUNNING}

// -------------- class SetTextField ------------------------
private class SetTextField implements Runnable
{
  private String text;

  SetTextField(String text)
  {
    this.text = text;
  }

  public void run()
  {
    textFieldSteps.setText(text);
  }
}

// -------------- class ButtonActionAdapter -----------------
private class ButtonActionAdapter implements ActionListener
{
  public void actionPerformed(ActionEvent evt)
  {
    if (evt.getSource() == goButton)
      goButton_actionPerformed();
    if (evt.getSource() == closeButton)
      closeButton_actionPerformed();
  }
}

// -------------- Application class -------------------------
  private JFrame frame;
  private JPanel controlPane = new JPanel();
  private JLabel labelSteps =
    new JLabel("Number of Steps (1..20): ");
  private JTextField textFieldSteps = new JTextField("10", 3);
  private JButton goButton = new JButton("Go");
  private JButton closeButton = new JButton("Close");
  private volatile State state = State.IDLE;
  private volatile int nbSteps;
  private volatile int step;

  public SwingEx2()
  {
    createGui();

    // Event loop
    while (true)
```

```
    {
      switch (state)
      {
        case IDLE:
           delay(1);
           break;

        case RUNNING:
          System.out.println(
            "Working hard step # " + (step+1));
          delay(500); // Do something more intelligent...
          SwingUtilities.invokeLater(
            new SetTextField("" + (step+1)));
          step++;
          if (step == nbSteps)
            state = state.IDLE;
          break;
      }
    }
  }

  private void createGui()
  {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                          WindowConstants.EXIT_ON_CLOSE);
    init();
    frame.pack();
    frame.setVisible(true);
  }

  private void init()
  {
    controlPane.add(labelSteps);
    controlPane.add(textFieldSteps);
    controlPane.add(goButton);
    controlPane.add(closeButton);
    frame.getContentPane().add(controlPane);
    goButton.addActionListener(new ButtonActionAdapter());
    closeButton.addActionListener(new ButtonActionAdapter());
  }

  private void goButton_actionPerformed()
  {
    if (state == state.IDLE)
    {
      try
      {
```

```
        String steps = textFieldSteps.getText().trim();
        nbSteps = Integer.parseInt(steps);
        if (nbSteps < 1 || nbSteps > 20)
          throw new NumberFormatException();
        step = 0;
        state = state.RUNNING;
      }
      catch (NumberFormatException ex)
      {
        JOptionPane.showMessageDialog(
               null, "Sorry, invalid number of steps." +
              "\nPlease enter a number in the range 1..20");
      }
    }
  }

  private void closeButton_actionPerformed()
  {
    System.exit(0);
  }

  private void delay(int ms)
  {
    try
    {
      Thread.currentThread().sleep(ms);
    }
    catch (InterruptedException ex) {}
  }

  public static void main(String[] args)
  {
    new SwingEx2();
  }
}
```

3.3.3 Resultat

Das Programm erfüllt die Vorgaben und entspricht den beiden Swing-Regeln. Für die Programmierausbildung heikel ist aber, dass bereits für eine derart einfache Programmieraufgabe fortgeschrittene Kenntnisse von Java vorausgesetzt werden.

3.3.4 Kommentar

Da wir für die Verwendung von `JTextField.setText()` eine neue Klasse erzeugen müssen, können wir diese gerade so erweitern, dass `setText()` als Methode dieser Klasse aufgerufen werden kann. Dazu vererben wir `JTextField` in der Klasse `_JTextComponent` und überschreiben dort `setText()` derart, dass die gleichnamige Methode der Superklasse mit `invokeAndWait()` zur Ausführung gelangt.

```
// _JTextField.java

import javax.swing.*;

public class _JTextField
  extends JTextField implements Runnable
{
  private String text;

  public _JTextField(String text, int columns)
  {
    super(text, columns);
  }

  public void setText(String text)
  {
    this.text = text;
    try
    {
      SwingUtilities.invokeAndWait(this);
    }
    catch (Exception ex) {}
  }

  public void run()
  {
    assert SwingUtilities.isEventDispatchThread() :
       "Not in EDT";
    super.setText(text);
  }
}
```

Statt den Haupt-Thread mit der Durchführung der Simulation zu beauftragen, kann auch ein neuer User-Thread erzeugt werden. Dieses Vorgehen ist sogar vorzuziehen, damit sich eine Entflechtung des Codes ergibt. Der neue Thread wird oft *Worker* genannt, da er die länger dauernde *Arbeit* übernimmt.

```
// SwingEx3.java
// User thread, compiles V1.5 up
```

```
import java.awt.event.*;
import javax.swing.*;
import javax.swing.text.*;

public class SwingEx3
{

enum State {IDLE, RUNNING}

// -------------- class Worker -----------------------------
class Worker extends Thread
{
  private int nbSteps;

  Worker(int nbSteps)
  {
    this.nbSteps = nbSteps;
  }

  void terminate()
  {
    state = state.IDLE;
    try
    {
      join(); // Wait until worker has finished
    }
    catch (InterruptedException ex) {}
  }

  public void run()
  {
    int step = 0;
    while (state == state.RUNNING)
    {
       System.out.println("Working hard step # " + (step+1));
       delay(500); // Do something more intelligent...
       textFieldSteps.setText("" + (step+1));
       step++;
       if (step == nbSteps)
         state = state.IDLE;
    }
  }

  private void delay(int ms)
  {
    try
    {
      Thread.currentThread().sleep(ms);
```

```
    }
    catch (InterruptedException ex) {}
  }
}

// -------------- class ButtonActionAdapter -----------------
private class ButtonActionAdapter implements ActionListener
{
  public void actionPerformed(ActionEvent evt)
  {
    if (evt.getSource() == goButton)
      goButton_actionPerformed();
    if (evt.getSource() == closeButton)
      closeButton_actionPerformed();
  }
}

// -------------- Application class -------------------------
  private JFrame frame;
  private JPanel controlPane = new JPanel();
  private JLabel labelSteps =
    new JLabel("Number of Steps (1..20): ");
  private
    _JTextField textFieldSteps = new _JTextField("10", 3);
  private JButton goButton = new JButton("Go");
  private JButton closeButton = new JButton("Close");
  // Worker thread
  Worker worker = null;
  // Interthread communication
  private volatile State state = State.IDLE;

  public SwingEx3()
  {
    createGui();
    int step = 0;
    int nbSteps = 0;
    // main-thread will terminate now
  }

  private void createGui()
  {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                          WindowConstants.EXIT_ON_CLOSE);
    init();
    frame.pack();
    frame.setVisible(true);
  }
```

```
  private void init()
  {
    controlPane.add(labelSteps);
    controlPane.add(textFieldSteps);
    controlPane.add(goButton);
    controlPane.add(closeButton);
    frame.getContentPane().add(controlPane);
    goButton.addActionListener(new ButtonActionAdapter());
    closeButton.addActionListener(new ButtonActionAdapter());
  }

  private void goButton_actionPerformed()
  {
    if (state == state.IDLE)
    {
      try
      {
        String steps = textFieldSteps.getText().trim();
        int nbSteps = Integer.parseInt(steps);
        if (nbSteps < 1 || nbSteps > 20)
          throw new NumberFormatException();
        state = state.RUNNING;
        worker = new Worker(nbSteps);
        worker.start();
      }
      catch (NumberFormatException ex)
      {
        JOptionPane.showMessageDialog(
               null, "Sorry, invalid number of steps." +
              "\nPlease enter a number in the range 1..20");
      }
    }
  }

  private void closeButton_actionPerformed()
  {
    System.exit(0);
  }

  public static void main(String[] args)
  {
    new SwingEx3();
  }
}
```

3.3.5 Resultat

Das Programm läuft korrekt, allerdings fehlt uns immer noch die Möglichkeit, eine laufende Simulation mit einem Stop-Button anzuhalten.

3.3.6 Kommentar

Beim Beenden mit dem Close-Button der Titelleiste oder mit `System.exit()` wird ebenfalls der User-Thread beendet. Wir benötigen daher die dafür vorgesehene Methode `terminate()` in unserem Programm nicht. Diese ist so geschrieben, dass sie zuerst durch das Setzen der state-Variablen die run-Methode zu Ende laufen lässt und nachher mit `join()` auf das Ende des Threads wartet.

Sun ist sich sehr wohl der zusätzlichen Schwierigkeiten bewusst, die durch ein striktes Einhalten der beiden Swing-Regeln entstehen. Aus diesem Grund kann man von Sun's Website eine Hilfsklasse `SwingWorker` herunterladen, welche das Problem etwas entschärft. Im folgenden Programm wird diese Klasse verwendet und gleichzeitig noch das ursprüngliche Ziel erreicht, während des Simulationsdurchlaufs den Go-Button in einen Stop-Button zu verwandeln, mit dem man die Simulation anhalten kann. Wir deklarieren dazu eine Methode `setControls()` welche die Komponenten unter Verwendung des aktuellen Zustands korrekt darstellt. Dies umfasst gewöhnlich auch ihre Aktivierung bzw. Deaktivierung (graying), wozu üblicherweise `setEnabled()` verwendet wird. Für das Einhalten der Einzelthread-Regel ist es wichtig, dass `setControls()` ausschließlich im EDT aufgerufen wird.

Die Klasse `SwingWorker` ist abstrakt, man muss daher daraus eine eigene Klasse ableiten, die wir `MySwingWorker` nennen. Damit sie problemlos auf die Instanzvariablen der Applikationsklasse zugreifen kann, deklarieren wir sie als innere Klasse. In `MySwingWorker` überschreibt man die Methoden `construct()` und `finished()`. Beim Aufruf der Methode `start()` wird `construct()` in einem zum `SwingWorker` gehörenden Thread aufgerufen. Läuft die Methode `construct()` zu Ende, so wird `finished()` im EDT ausgeführt. In `construct()` führt man daher die Berechnungen durch, in `finished()` hingegeben die Rückmeldungen an das GUI. Zudem kann `construct()` als return-Wert noch eine Objektreferenz zurückgeben, die man mit `get()` abholen kann.

In unserem Fall müssen wir bei jedem Schritt einen neuen SwingWorker erzeugen, damit wir die aktuelle Schrittzahl im Textfeld anzeigen können. Aus diesem Grund instanzieren wir in `finished()` solange einen neuen SwingWorker, bis alle Schritte durchlaufen sind, was zu einer etwas ungewöhnlichen rekursiven Programmstruktur führt.

```
// SwingEx4.java
// Uses Sun's class SwingWorker (pre Mustang version)
// Download source code from http://java.sun.com

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
public class SwingEx4
{

enum State {IDLE, RUNNING}

// -------------- class MySwingWorker -----------------------
private class MySwingWorker extends SwingWorker
{
  private int nbSteps;
  private int step;

  MySwingWorker(int nbSteps, int step)
  {
    this.nbSteps = nbSteps;
    this.step = step;
  }

  // Override construct()
  public Object construct()
  {
    System.out.println("working hard # " + step);
    sleep(500); // Do something more intelligent...
    return null;
  }

  // Override finished(), runs in EDT
  public void finished()
  {
   textFieldSteps.setText("" + step);
   if (step == nbSteps || state == State.IDLE)
   {
     state = state.IDLE;
     setControls();
   }
   else
     new MySwingWorker(nbSteps, step + 1).start();
  }

  private void sleep(int ms)
  {
    try
    {
      Thread.currentThread().sleep(ms);
    }
    catch (InterruptedException ex) {}
  }
}
```

```
// -------------- class ButtonActionAdapter -----------------
private class ButtonActionAdapter implements ActionListener
{
  public void actionPerformed(ActionEvent evt)
  {
    if (evt.getSource() == goButton)
      goButton_actionPerformed();
    if (evt.getSource() == closeButton)
      closeButton_actionPerformed();
  }
}

// -------------- Application class ------------------------
  private JFrame frame;
  private JPanel controlPane = new JPanel();
  private JLabel labelSteps =
    new JLabel("Number of Steps (1..20): ");
  private JTextField textFieldSteps = new JTextField("10", 3);
  private JButton goButton = new JButton("Go");
  private JButton closeButton = new JButton("Close");
  // Interthread communication
  private volatile State state = State.IDLE;

  public SwingEx4()
  {
    createGui();
    int step = 0;
    int nbSteps = 0;
    // main-thread will terminate
  }

  private void createGui()
  {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                          WindowConstants.EXIT_ON_CLOSE);
    init();
    frame.pack();
    frame.setVisible(true);
  }

  private void init()
  {
    controlPane.add(labelSteps);
    controlPane.add(textFieldSteps);
    controlPane.add(goButton);
    controlPane.add(closeButton);
```

```
    frame.getContentPane().add(controlPane);
    goButton.addActionListener(new ButtonActionAdapter());
    closeButton.addActionListener(new ButtonActionAdapter());
    goButton.setPreferredSize(new Dimension(70, 20));
    closeButton.setPreferredSize(new Dimension(70, 20));
   }

  private void goButton_actionPerformed()
  {
    switch (state)
    {
      case IDLE:
        try
        {
          String steps = textFieldSteps.getText().trim();
          int nbSteps = Integer.parseInt(steps);
          if (nbSteps < 1 || nbSteps > 20)
            throw new NumberFormatException();
          state = state.RUNNING;
          setControls();
          new MySwingWorker(nbSteps, 1).start();
        }
        catch (NumberFormatException ex)
        {
          JOptionPane.showMessageDialog(
                 null, "Sorry, invalid number of steps." +
                "\nPlease enter a number in the range 1..20");
        }
        break;

      case RUNNING:
        state = state.IDLE;
        setControls();
        break;
    }
  }

  private void closeButton_actionPerformed()
  {
    // Do some cleanup here
    System.exit(0);
  }

  private void setControls()
  {
    switch (state)
    {
      case RUNNING:
```

```
        goButton.setText("Stop");
        break;

      case IDLE:
        goButton.setText("Go");
        break;
    }
  }

  public static void main(String[] args)
  {
    new SwingEx4();
  }
}
```

3.3.7 Resultat

Das Programm läuft wie erwartet, verglichen mit dem vorher selbst deklarierten `Worker` ist der Code nicht wesentlich einfacher geworden.

3.3.8 Kommentar

Die ungewohnte rekursive Struktur ließe sich vermeiden, falls man vom `SwingWorker` Zwischenresultate zurück erhalten könnte. In der neusten Java-Version 6 (Mustang) ist tatsächlich im JDK eine Klasse `javax.swing.SwingWorker` integriert, die Zwischenwerte zurück melden kann. Dazu werden sogar zwei Verfahren angeboten: Zwischenwerte können mit `publish()` vom Worker-Thread in eine Queue gesendet werden, worauf sie asynchron mit der Methode `process()`, die im EDT läuft, zurück geholt und verarbeiten werden können. Zwischenwerte können aber auch einen Event "*feuern*", und in einem registrierten Listener in Empfang genommen werden. Die neue SwingWorker-Klasse ist zudem durch Generics typensicherer geworden. Es gibt folgende ungefähren Entsprechungen:

Alter SwingWorker	**javax.swing.SwingWorker**
construct()	doInBackground()
finished()	done()
start()	execute()
getValue()	get()
interrupt()	cancel()
-	publish()

-	process()
-	addPropertyChangeListener()
-	firePropertyChange()

Im folgenden Beispiel wird der aktuelle Wert von `step` innerhalb von `doInBackground()` mit `publish()` gespeichert. Dadurch wird die Notifikations-Methode `process()` getriggert, in der man die gespeicherten Werte abholen und Swing-Methoden direkt übergeben kann. Dabei ist wichtig, dass nicht jeder Aufruf von `publish()` zu einem Aufruf von `process()` führen muss. Es ist dem EDT freigestellt, wann er `process()` aufrufen will. Es können daher in einem einzigen Aufruf von `process()` mehrere vorher gespeicherte Werte bereit stehen. Die Implementierung erfolgt etwas ungewöhnlich mittels einer **variablen Argumentliste** (**varargs**).

Mit `isCancelled()` wird getestet, ob durch den Aufruf von `cancel()` in der Callback-Methode des Stop-Buttons die Simulation vorzeitig abgebrochen werden soll.

```
// SwingEx5.java
// Uses Sun's new API class javax.swing.SwingWorker
// Needs J2SE V6 (Mustang)
// Notification by publish/process

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class SwingEx5
{

enum State {IDLE, RUNNING}

// -------------- class MySwingWorker -----------------------
private class MySwingWorker
  extends javax.swing.SwingWorker<Object, Integer>
{
  int nbSteps;

  MySwingWorker(int nbSteps)
  {
    this.nbSteps = nbSteps;
  }

  @Override
  public Object doInBackground()
  {
    int step = 0;
    while (!isCancelled() && step < nbSteps)
```

```
    {
      System.out.println("working hard # " + (step+1));
      delay(500); // Do something more intelligent...
      publish(new Integer(step));
      step++;
    }
    return null;
  }

  @Override
  protected void process(Integer... steps)
  {
    for (int i : steps)
      textFieldSteps.setText("" + (i+1));
  }

  @Override
  protected void done()
  {
    state = State.IDLE;
    setControls();
  }

  private void delay(int ms)
  {
    try
    {
      Thread.currentThread().sleep(ms);
    }
    catch (InterruptedException ex) {}
  }
}


// -------------- class ButtonActionAdapter -----------------
private class ButtonActionAdapter implements ActionListener
{
  public void actionPerformed(ActionEvent evt)
  {
    if (evt.getSource() == goButton)
      goButton_actionPerformed();
    if (evt.getSource() == closeButton)
      closeButton_actionPerformed();
  }
}

// -------------- Application class -------------------------
  private JFrame frame;
```

```
  private JPanel controlPane = new JPanel();
  private JLabel labelSteps =
    new JLabel("Number of Steps (1..20): ");
  private JTextField textFieldSteps = new JTextField("10", 3);
  private JButton goButton = new JButton("Go");
  private JButton closeButton = new JButton("Close");
  private State state = State.IDLE;
  MySwingWorker worker;

  public SwingEx5()
  {
    createGui();
    int step = 0;
    int nbSteps = 0;
    // main-thread will terminate
  }

  private void createGui()
  {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                          WindowConstants.EXIT_ON_CLOSE);
    init();
    frame.pack();
    frame.setVisible(true);
  }

  private void init()
  {
    controlPane.add(labelSteps);
    controlPane.add(textFieldSteps);
    controlPane.add(goButton);
    controlPane.add(closeButton);
    frame.getContentPane().add(controlPane);
    goButton.addActionListener(new ButtonActionAdapter());
    closeButton.addActionListener(new ButtonActionAdapter());
    goButton.setPreferredSize(new Dimension(70, 20));
    closeButton.setPreferredSize(new Dimension(70, 20));
   }

  private void goButton_actionPerformed()
  {
    switch (state)
    {
      case IDLE:
        try
        {
          String steps = textFieldSteps.getText().trim();
```

```
          int nbSteps = Integer.parseInt(steps);
          if (nbSteps < 1 || nbSteps > 20)
            throw new NumberFormatException();
          state = state.RUNNING;
          setControls();
          worker = new MySwingWorker(nbSteps);
          worker.execute();
        }
        catch (NumberFormatException ex)
        {
          JOptionPane.showMessageDialog(
                 null, "Sorry, invalid number of steps." +
                "\nPlease enter a number in the range 1..20");
        }
        break;

      case RUNNING:
        worker.cancel(true);
        break;
    }
  }

  private void closeButton_actionPerformed()
  {
    System.exit(0);
  }

  private void setControls()
  {
    switch (state)
    {
      case RUNNING:
        goButton.setText("Stop");
        break;

      case IDLE:
        goButton.setText("Go");
        break;
    }
  }

  public static void main(String[] args)
  {
    new SwingEx5();
  }
}
```

3.3.9 Resultat

Das Programm läuft korrekt, der Aufwand an zusätzlichem Know-How ist aber beträchtlich. Bei

```
@Override
```

handelt es sich um eine **Anotation**, die garantiert, dass die nachfolgend deklarierte Methode tatsächlich eine Methode der Basisklasse überschreibt.

3.3.10 Kommentar

Elegant ist die Möglichkeit, mit dem neuen `SwingWorker` Zwischenwerte über einen `PropertyChangeEvent` zurück zu geben, indem man einen `PropertyEventListener` registriert. Die Callback-Methode `propertyChange()` wird beim Feuern eines Events vom EDT aufgerufen und man kann aus dem Parameter den Wert einer beliebigen Property herausholen, der beim Feuern mit `firePropertyChange()` (und einigen anderen Methoden) gesetzt wird. Im Callback kann direkt eine Swing-Methode aufgerufen werden, da der Callback im EDT läuft. Im folgenden Beispiel wird die Property mit dem Namen *Step* auf den aktuellen Integer-Wert von step gesetzt.

```
// SwingEx6.java
// Uses Sun's new API class javax.swing.SwingWorker
// Needs J2SE V6 (Mustang)
// Notification with PropertyChangeListener

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.beans.*;

public class SwingEx6
{

enum State {IDLE, RUNNING}

// -------------- class MySwingWorker -----------------------
private class MySwingWorker
  extends javax.swing.SwingWorker<Object, Object>
{
  int nbSteps;

  MySwingWorker(int nbSteps)
  {
    this.nbSteps = nbSteps;
    addPropertyChangeListener(
      new PropertyChangeListener()
```

```
      {
        public void propertyChange(PropertyChangeEvent evt)
        {
          if ("Step".equals(evt.getPropertyName()))
            textFieldSteps.
              setText("" + (Integer)evt.getNewValue());
        }
      });
  }

  @Override
  public Object doInBackground()
  {
    int step = 0;
    while (!isCancelled() && step < nbSteps)
    {
      System.out.println("working hard # " + (step+1));
      delay(500); // Do something more intelligent...
      firePropertyChange("Step",
                         new Integer(-1),
                         new Integer(step+1));
      step++;
    }
    return null;
  }

  @Override
  protected void done()
  {
    state = State.IDLE;
    setControls();
  }

  private void delay(int ms)
  {
    try
    {
      Thread.currentThread().sleep(ms);
    }
    catch (InterruptedException ex) {}
  }
}


// -------------- class ButtonActionAdapter -----------------
private class ButtonActionAdapter implements ActionListener
{
  public void actionPerformed(ActionEvent evt)
```

```
  {
    if (evt.getSource() == goButton)
      goButton_actionPerformed();
    if (evt.getSource() == closeButton)
      closeButton_actionPerformed();
  }
}

// -------------- Application class ------------------------
  private JFrame frame;
  private JPanel controlPane = new JPanel();
  private JLabel labelSteps =
    new JLabel("Number of Steps (1..20): ");
  private JTextField textFieldSteps = new JTextField("10", 3);
  private JButton goButton = new JButton("Go");
  private JButton closeButton = new JButton("Close");
  private State state = State.IDLE;
  MySwingWorker worker;

  public SwingEx6()
  {
    createGui();
    int step = 0;
    int nbSteps = 0;
    // main-thread will terminate
  }

  private void createGui()
  {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                          WindowConstants.EXIT_ON_CLOSE);
    init();
    frame.pack();
    frame.setVisible(true);
  }

  private void init()
  {
    controlPane.add(labelSteps);
    controlPane.add(textFieldSteps);
    controlPane.add(goButton);
    controlPane.add(closeButton);
    frame.getContentPane().add(controlPane);
    goButton.addActionListener(new ButtonActionAdapter());
    closeButton.addActionListener(new ButtonActionAdapter());
    goButton.setPreferredSize(new Dimension(70, 20));
    closeButton.setPreferredSize(new Dimension(70, 20));
```

```
 }

private void goButton_actionPerformed()
{
  switch (state)
  {
    case IDLE:
      try
      {
        String steps = textFieldSteps.getText().trim();
        int nbSteps = Integer.parseInt(steps);
        if (nbSteps < 1 || nbSteps > 20)
          throw new NumberFormatException();
        state = state.RUNNING;
        setControls();
        worker = new MySwingWorker(nbSteps);
        worker.execute();
      }
      catch (NumberFormatException ex)
      {
        JOptionPane.showMessageDialog(
               null, "Sorry, invalid number of steps." +
              "\nPlease enter a number in the range 1..20");
      }
      break;

    case RUNNING:
      worker.cancel(true);
      break;
  }
}

private void closeButton_actionPerformed()
{
  System.exit(0);
}

private void setControls()
{
  switch (state)
  {
    case RUNNING:
      goButton.setText("Stop");
      break;

    case IDLE:
      goButton.setText("Go");
      break;
```

```
    }
  }

  public static void main(String[] args)
  {
    new SwingEx6();
  }
}
```

3.3.11 Kommentar

Da es sich um eine Property mit einem Integer-Wert handelt, könnte man statt `firePropertyChange()` auch einfacher `setProgress()` verwenden. Da damit die Property mit dem Namen *progress* gesetzt wird, müsste man den Wert in der Methode `propertyChange()` mit

```
if ("progress".equals(evt.getPropertyName()))
  textFieldSteps.setText("" + (Integer)evt.getNewValue());
```

zurück holen.