Leider doch. Es GIBT eine Kehrseite.

Threads können zu Nebenläufigkeitsproblemen führen.

Nebenläufigkeitsprobleme führen zu Konkurrenzsituationen. Konkurrenzsituationen führen zu Datenverfälschung. Verfälschte Daten lösen Ängste aus ... den Rest kennen Sie ja.

Es läuft alles auf das folgende, potenziell tödliche Szenario hinaus: Zwei oder mehr Threads haben Zugriff auf die *Daten* desselben Objekts. Mit anderen Worten, Methoden auf zwei verschiedenen Stacks rufen beide z.B. Getter und Setter ein und desselben Objekts auf dem Heap auf.

Es ist eine »Die-linke-Hand-weiß-nicht-was-die-rechte-tut«-Geschichte. Zwei Threads, sorgenfrei und unbeschwert, führen fröhlich pfeifend ihre Methoden aus, und jeder von ihnen glaubt, er sei »der einzig wahre Thread«. Der einzige, der zählt. Denn ein Thread, der gerade nicht läuft, sondern im Zustand »lauffähig« (oder »blockiert«) ist, ist ja praktisch bewusstlos. Wenn er wieder zum laufenden Thread wird, weiß er gar nicht, dass er jemals angehalten hat.

Ehe in Gefahr

Ist dieses Paar noch zu retten?

Gleich – in einer Sondersendung aus unserer Reihe »Fragen Sie Dr. Winter«!

[Abschrift von Folge 42]

Herzlich willkommen in unserer Sendung!

In unserem aktuellen Thema geht es um die beiden wichtigsten Gründe, warum Paare sich trennen – Finanz- und Schlafprobleme.

Rainer und Monika: Opfer des »Zwei-Leute-ein-Konto«-Problems.

Unser heutiges Problempaar, Rainer und Monika, teilt Bankkonto und Bett. Allerdings nicht mehr lange, wenn wir nicht eine Lösung finden können. Das Problem? Die klassische »Zwei-Leute-ein-Bankkonto«-Geschichte. Und so hat Monika es mir beschrieben:

»Rainer und ich haben vereinbart, dass keiner von uns das Konto überziehen wird. Wir machen es also so: Wer von uns etwas abheben will, muss den Kontostand überprüfen, bevor er die Abhebung durchführt. Es schien alles so einfach. Aber plötzlich werden unsere Schecks gesperrt, und wir zahlen jede Menge Überziehungszinsen!

Ich dachte, das wäre unmöglich, unser Verfahren wäre sicher. Aber dann passierte *das hier*:

Rainer brauchte 50 €, also hat er den Kontostand geprüft und gesehen, dass noch 100 € drauf waren. Kein Problem also, er will den Betrag abheben. **Aber dann schläft er erst einmal ein!**

Rainer schläft ein, nachdem er den Kontostand geprüft hat, aber bevor er das Geld abgehoben hat. Wenn er aufwacht, hebt er es sofort ab, ohne sich den Kontostand noch einmal anzusehen.

Und jetzt komme ich, während Rainer noch schläft, und will eine Abhebung von 100 € machen. Ich prüfe den Kontostand; es sind 100 € drauf (weil Rainer noch schläft und seinen Betrag noch nicht abgehoben hat), also denke ich, kein Problem. Ich hebe das Geld ab, und da ist noch alles okay. Aber dann wacht Rainer auf, führt seine Abhebung zu Ende, und plötzlich ist unser Konto überzogen! Er wusste noch nicht mal, dass er eingeschlafen war, also hat er einfach weitergemacht und seinen Vorgang zu Ende geführt, ohne noch einmal den Kontostand zu überprüfen. Sie müssen uns helfen, Dr. Winter!«

Gibt es eine Lösung? Oder ist die Ehe dem Untergang geweiht? Wir können Rainer nicht am Einschlafen hindern, aber können wir vielleicht sicherstellen, dass Monika nicht auf das Konto zugreifen kann, bevor er aufgewacht ist?

Nehmen Sie sich einen Moment Zeit und denken Sie darüber nach, während wir eine Werbepause machen.

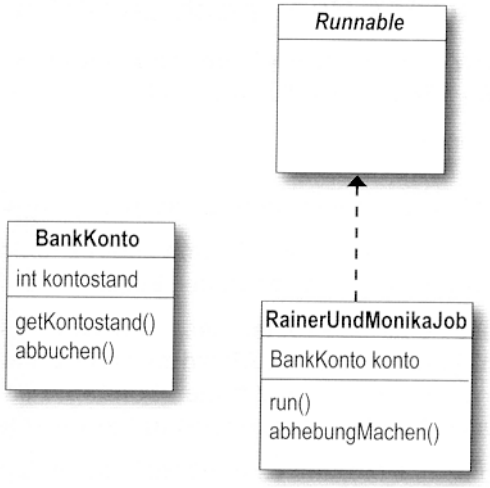
Buch „Java von Kopf bis Fuß“ – 505

Das Rainer-und-Monika-Problem in Codeform

Das folgende Beispiel zeigt, was passieren kann, wenn zwei Threads (Rainer und Monika) ein Objekt (das Bankkonto) miteinander teilen.

Der Code hat zwei Klassen, BankKonto und RainerUndMonikaJob. Die Klasse RainerUndMonikaJob implementiert Runnable und repräsentiert das Verhalten, das sowohl Rainer als auch Monika haben: den Kontostand prüfen und die Abhebung durchführen. Aber natürlich schläft jeder Thread zwischen dem Überprüfen des Kontostands und dem eigentlichen Abheben ein.

Die Klasse RainerUndMonikaJob hat eine Instanzvariable vom Typ BankKonto, die das gemeinsame Konto repräsentiert.



❶ Eine Instanz von RainerUndMonikaJob machen.

Das RainerUndMonikaJob-Objekt ist das Runnable (der Job), und da Monika und Rainer das Gleiche tun (Kontostand prüfen und Geld abheben), brauchen wir nur eine einzige Instanz.

```
RainerUndMonikaJob derJob = new RainerUndMonikaJob();
```

❷ Zwei Threads mit dem gleichen Runnable machen

(der RainerUndMonikaJob-Instanz).

```
Thread eins = new Thread(derJob);
Thread zwei = new Thread(derJob);
```

❸ Den Threads einen Namen geben und sie starten.

```
eins.setName("Rainer");
zwei.setName("Monika");
eins.start();
zwei.start();
```

❹ Beide Threads bei der Ausführung der run()-Methode beobachten

(bei der Überprüfung des Kontostands und dem eigentlichen Abheben).

Ein Thread repräsentiert Rainer, der andere repräsentiert Monika. Beide Threads prüfen immer wieder den Kontostand und heben dann etwas ab - aber nur, wenn genug da ist!

```
if (konto.getKontostand() >= betrag) {
  try {
    Thread.sleep(500);
  } catch(InterruptedException ex) {
    ex.printStackTrace();
  }
}
```

In der run()-Methode machen wir genau das, was auch Monika und Rainer tun würden – den Kontostand prüfen und Geld abheben, wenn genug auf dem Konto ist.

Dies sollte vor der Überziehung des Kontos schützen.

Außer ... wenn Monika und Rainer jedes Mal einschlafen, nachdem sie den Kontostand geprüft haben, aber bevor sie die Abhebung zu Ende führen.

Das Rainer-und-Monika-Beispiel

```
class BankKonto {
  private int kontostand = 100;

  public int getKontostand() {
    return kontostand;
  }

  public void abbuchen(int betrag) {
    kontostand = kontostand - betrag;
  }
}

public class RainerUndMonikaJob implements Runnable {

  private BankKonto konto = new BankKonto();

  public static void main (String [] args) {
    RainerUndMonikaJob derJob = new RainerUndMonikaJob();
    Thread eins = new Thread(derJob);
    Thread zwei = new Thread(derJob);
    eins.setName("Rainer");
    zwei.setName("Monika");
    eins.start();
    zwei.start();
  }

  public void run() {
    for (int x = 0; x < 10; x++) {
      abhebungMachen(10);
      if (konto.getKontostand() < 0) {
        System.out.println("Überzogen!");
      }
    }
  }

  private void abhebungMachen(int betrag) {
    if (konto.getKontostand() >= betrag) {
      System.out.println(Thread.currentThread().getName() + " will abheben.");
      try {
        System.out.println(Thread.currentThread().getName() + " schläft ein.");
        Thread.sleep(500);
      } catch(InterruptedException ex) {ex.printStackTrace(); }
      System.out.println(Thread.currentThread().getName() + " ist aufgewacht.");
      konto.abbuchen(betrag);
      System.out.println(Thread.currentThread().getName() + " führt die Abhebung zu Ende.");
    }
    else {
      System.out.println("Leider nicht genug Geld für " + Thread.currentThread().getName());
    }
  }
}
```

Zu Beginn beträgt der Kontostand 100 €.

Es gibt nur EINE Instanz des RainerUndMonika-Job, d.h. nur EINE Instanz des Bankkontos. Beide Threads werden auf dieses eine Konto zugreifen.

Instantiieren Sie das Runnable (den Job).

Machen Sie zwei Threads und übergeben Sie beiden den gleichen Runnable-Job. Das bedeutet, dass beide Threads auf ein und dieselbe Instanzvariable konto in der Runnable-Klasse zugreifen.

In der run()-Methode durchläuft ein Thread die Schleife und versucht bei jeder Iteration, eine Abhebung zu machen. Nach dem Abheben prüft er den Kontostand erneut, um zu sehen, ob das Konto überzogen ist.

Prüfen Sie den Kontostand, und wenn nicht genug Geld da ist, wird einfach eine Meldung ausgegeben. WENN genug da ist, schlafen wir ein, wachen dann wieder auf und führen die Abhebung zu Ende, genau wie Rainer es gemacht hat.

Wir haben eine ganze Menge print-Anweisungen eingefügt, damit wir bei der Ausführung sehen, was passiert.

```
Datei  Bearbeiten  Fenster  Hilfe  KreditKarte
Rainer ist im Begriff abzuheben.
Rainer schläft ein.
Monika ist aufgewacht.
Monika führt die Abhebung zu Ende.
Monika ist im Begriff abzuheben.
Monika schläft ein.
Rainer ist aufgewacht.
Rainer führt die Abhebung zu Ende.
Rainer ist im Begriff abzuheben.
Rainer schläft ein.
Monika ist aufgewacht.
Monika führt die Abhebung zu Ende.
Monika ist im Begriff abzuheben.
Monika schläft ein.
Rainer ist aufgewacht.
Rainer führt die Abhebung zu Ende.
Rainer ist im Begriff abzuheben.
Rainer schläft ein.
Monika ist aufgewacht.
Monika führt die Abhebung zu Ende.
Leider nicht genug Geld für Monika
Leider nicht genug Geld für Monika
Leider nicht genug Geld für Monika
Leider nicht genug Geld für Monika
Leider nicht genug Geld für Monika
Rainer ist aufgewacht.
Rainer führt die Abhebung zu Ende.
Überzogen!
Leider nicht genug Geld für Rainer
Überzogen!
Leider nicht genug Geld für Rainer
Überzogen!
Leider nicht genug Geld für Rainer
Überzogen!
```

Wie konnte das passieren?

Die Methode abhebungMachen() überprüft jedes Mal den Kontostand, bevor die eigentliche Abhebung durchgeführt wird – und trotzdem überziehen wir das Konto.

Hier ein mögliches Szenario:

Rainer prüft den Kontostand, sieht, dass genug Geld da ist, und schläft dann ein.

In der Zwischenzeit kommt Monika und überprüft den Kontostand. Sie sieht ebenfalls, dass genug Geld da ist, und hat keine Ahnung, dass Rainer gleich aufwachen wird und eine Abhebung zu Ende führt.

Monika schläft ein.

Rainer wacht auf und führt seine Abhebung zu Ende.

Monika wacht auf und führt ihre Abhebung zu Ende. Und da haben wir den Salat! In der Zwischenzeit - nachdem sie den Kontostand geprüft und bevor sie die Abhebung zu Ende geführt hatte - war Rainer aufgewacht und hatte Geld vom Konto genommen.

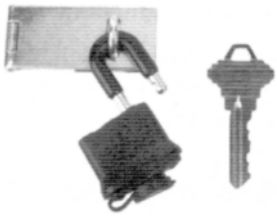
Monikas Überprüfung des Kontostands war nicht gültig, weil Rainer ihn bereits überprüft hatte und noch mitten in einem Abhebungsvorgang steckte.

Monika darf nicht an das Konto gehen können, bevor Rainer aufwacht und seine Transaktion zu Ende führt. Und umgekehrt.

Sie brauchen ein Schloss für den Kontozugang!

Das Schloss funktioniert so:

❶ Zu der Kontotransaktion (Kontostand prüfen und Geld abheben) gibt es ein Schloss. Es existiert nur *ein* Schlüssel, und der hängt neben dem Schloss, bis jemand auf das Konto zugreifen will.



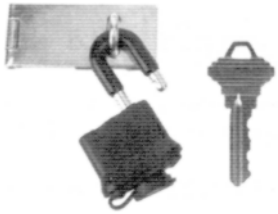
Die Bankkonto-Transaktion ist unversperrt, wenn gerade niemand auf das Konto zugreift.

❷ Wenn Rainer auf das Konto zugreifen will (um den Kontostand zu prüfen und Geld abzuheben), schließt er ab und steckt den Schlüssel in seine Hosentasche. Jetzt kann niemand anderes auf das Konto zugreifen, weil der Schlüssel weg ist.

Wenn Rainer auf das Konto zugreifen will, versperrt er das Schloss und nimmt den Schlüssel an sich.

❸ **Rainer behält den Schlüssel in seiner Tasche, bis er mit der Transaktion fertig ist.** Er hat den einzigen Schlüssel, daher kann Monika nicht auf das Konto (oder das Scheckbuch) zugreifen, bevor Rainer wieder aufgeschlossen und den Schlüssel zurückgegeben hat.

Wenn Rainer jetzt einschläft, nachdem er das Konto geprüft hat, hat er die Gewissheit, dass der Kontostand bei seinem Aufwachen noch der gleiche ist, weil er den Schlüssel behalten hat, während er schlief.



Wenn Rainer fertig ist, sperrt er das Schloss auf und hängt den Schlüssel zurück. Jetzt ist der Schlüssel für Monika (oder erneut für Rainer) verfügbar und gestattet den Zugriff auf das Konto.

Wir müssen die Methode abhebungMachen() als unteilbare atomare Einheit ausführen.

Wir müssen dafür sorgen, dass ein Thread, sobald er einmal in die Methode abhebungMachen() eingetreten ist, *diese Methode zu Ende ausführen darf*, bevor irgendein anderer Thread hineindarf.

Mit anderen Worten: Wir müssen sicherstellen, dass einem Thread, der den Kontostand geprüft hat, garantiert wird, dass er aufwachen und die Abhebung zu Ende ausführen kann, *bevor irgendein anderer Thread den Kontostand prüfen kann!*

Mit dem Schlüsselwort **`synchronized`** können Sie eine Methode so modifizieren, dass immer nur ein Thread auf einmal auf sie zugreifen kann.

Und so schützen Sie das Konto! Sie schließen nicht das Bankkonto selbst ab, sondern Sie schließen die Methode für die Kontotransaktion ab. Auf diese Weise kann ein Thread die gesamte Transaktion von Anfang bis Ende durchführen, selbst wenn er mitten in der Methode einschläft!

Wenn man also nicht das Bankkonto abschließt, was genau *wird* dann abgeschlossen? Das Runnable-Objekt? Der Thread selbst?

Das sehen wir uns auf der nächsten Seite an. Im Code ist es allerdings ganz einfach – fügen Sie in Ihrer Methodendeklaration einfach den Modifizierer `synchronized` ein:

Das Schlüsselwort `synchronized` bedeutet, dass ein Thread einen Schlüssel braucht, um auf den synchronisierten Code zugreifen zu können.

Um Ihre Daten (z.B. das Bankkonto) zu schützen, synchronisieren Sie die *Methoden*, die auf diese Daten einwirken.

```
private synchronized void abhebungMachen(int betrag) {

  if (konto.getKontostand() >= betrag) {
    System.out.println(Thread.currentThread().getName() + " will abheben.");
    try {
      System.out.println(Thread.currentThread().getName() + " schläft ein.");
      Thread.sleep(500);
    } catch(InterruptedException ex) {ex.printStackTrace(); }
    System.out.println(Thread.currentThread().getName() + " ist aufgewacht.");
    konto.abbuchen(betrag);
    System.out.println(Thread.currentThread().getName() + " führt die Abhebung zu Ende.");
  }
  else {
    System.out.println("Leider nicht genug Geld für " + Thread.currentThread().getName());
  }
}
```

(Anmerkung für Leser, die Ahnung von Physik haben: Ja, die Konvention für die Verwendung des Worts >>atomar<< in diesem Zusammenhang passt nicht zu der ganzen Sache mit den subatomaren Teilchen. Denken Sie an Newton und nicht an Einstein, wenn Sie das Wort >>atomar<< im Zusammenhang mit Threads oder Transaktionen hören. Diese Konvention stammt ja nicht von UNS. Wenn WIR zuständig wären, würden wir auf praktisch alles, was mit Threads zu tun hat, die Heisenbergsche Unschärferelation anwenden.)

Das Schloss eines Objekts verwenden

Jedes Objekt hat ein Schloss (im Deutschen häufiger als Sperre oder mit dem englischen Begriff *Lock* bezeichnet). Meistens ist das Schloss offen, dann können Sie sich einen daneben liegenden virtuellen Schlüssel dazudenken. Objektsperren kommen nur dann ins Spiel, wenn es synchronisierte Methoden gibt. Hat ein Objekt eine oder mehrere synchronisierte Methoden, ***kann ein Thread erst dann in eine synchronisierte Methode eintreten, wenn er den Schlüssel zum Schloss des Objekts erhält!***

Die Sperren gehören nicht zu einer *Methode*, sondern zum *Objekt*. Wenn ein Objekt zwei synchronisierte Methoden hat, bedeutet das nicht einfach, dass zwei Threads nicht gleichzeitig in dieselbe Methode eintreten können. Es bedeutet, dass zwei Threads nicht gleichzeitig in irgendeine der Methoden eintreten können.

Überlegen Sie mal: Wenn Sie mehrere Methoden haben, die auf die Instanzvariablen eines Objekts einwirken können, müssen alle diese Methoden durch »synchronized« geschützt werden.

Das Ziel der Synchronisierung ist der Schutz von kritischen Daten. Aber denken Sie daran, Sie verschließen nicht die Daten selbst, sondern synchronisieren die Methoden, die auf die Daten *zugreifen*.

Was passiert also, wenn ein Thread sich durch seinen Aufruf-Stack ackert (beginnend mit der run()-Methode) und dann plötzlich auf eine synchronisierte Methode stößt? Er erkennt, dass er einen Schlüssel für das Objekt braucht, bevor er Zutritt zu der Methode erhält. Er sieht sich nach dem Schlüssel um (um all das kümmert sich die JVM; es gibt in Java keine API, mit der man auf Objektschlösser zugreifen kann), und sobald dieser verfügbar ist, schnappt er ihn sich und tritt in die Methode ein.

Von diesem Zeitpunkt an klammert sich der Thread an diesen Schlüssel, als hinge sein Leben davon ab. Er lässt ihn nicht los, bevor er nicht mit der synchronisierten Methode fertig ist. Während dieser Thread also den Schlüssel festhält, erhält kein anderer Thread Zutritt zu *irgendeiner* der synchronisierten Methoden in diesem Objekt, weil der einzige Schlüssel für dieses Objekt nicht zur Verfügung steht.

Jedes Java-Objekt hat ein Schloss. Ein Schloss hat nur einen Schlüssel.

Meistens ist das Schloss unversperrt, und keiner interessiert sich dafür.

Aber wenn ein Objekt synchronisierte Methoden enthält, kann ein Thread nur dann in eine der synchronisierten Methoden eintreten, wenn der Schlüssel für das Schloss des Objekts zur Verfügung steht. Mit anderen Worten: nur wenn sich nicht bereits ein anderer Thread den einzigen Schlüssel geschnappt hat.

Das gefürchtete »Problem der verlorenen Aktualisierung«

Hier kommt ein weiteres klassisches Nebenläufigkeitsproblem, das aus der Datenbankwelt stammt. Es ist zwar eng mit der Rainer-und-Monika-Story verwandt, aber wir nehmen hier dieses Beispiel, um einige weitere Punkte aufzuzeigen.

Bei der verlorenen Aktualisierung (lost update) geht es um einen bestimmten Vorgang:

Schritt 1: Kontostand abfragen

```
int i = kontostand;
```

Schritt 2: Kontostand um 1 erhöhen

```
kontostand = i + 1;
```

Um das Problem aufzuzeigen, wenden wir einen Trick an: Wir zwingen den Computer, die Veränderung des Kontostands in zwei Schritten vorzunehmen. Im wahren Leben würde dafür eine einzige Anweisung genügen: **`kontostand++;`**

Aber dadurch, dass wir das in *zwei* Schritte aufteilen, wird das Problem eines nicht atomaren Vorgangs klar. Stellen Sie sich also vor, die zwei (oder mehr) Schritte dieser Methode bestünden nicht aus etwas so Trivialem wie »Kontostand abfragen und dann 1 zum aktuellen Stand addieren«, sondern sie wären komplexer und ließen sich nicht in einer Anweisung zusammenfassen.

Beim Problem der »verlorenen Aktualisierung« haben wir zwei Threads, die beide versuchen, den Kontostand zu erhöhen.

```
class TestSync implements Runnable {

  private int kontostand;

  public void run() {
    for(int i = 0; i < 50; i++) {
      inkrementieren();
      System.out.printin("kontostand beträgt " + kontostand);
    }
  }

  public void inkrementieren() {
    int i = kontostand;
    kontostand = i + 1;
  }
}
```

Jeder Thread läuft 50-mal und erhöht bei jedem Durchlauf den Kontostand.

Hier ist die kritische Stelle! Wir erhöhen den Kontostand, indem wir 1 zu dem Wert des Kontostands, den er ZUM ZEITPUNKT UNSERER ABFRAGE hatte, addieren (anstatt 1 zum AKTUELLEN Wert zu addieren).

```
public class TestSyncTest {
  public static void main (String[] args) {
    TestSync Job = new TestSync();
    Thread a = new Thread(job);
    Thread b = new Thread(job);
    a.start();
    b.start();
  }
}
```

Führen wir diesen Code aus ...

❶ Thread A läuft eine Weile

Setzen Sie den Wert des Kontostands in die Variable i.
Der Kontostand ist 0, also ist i jetzt gleich 0.
Setzen Sie den Wert des Kontostands auf das Ergebnis von i + 1.
Der Kontostand ist jetzt 1.
Setzen Sie den Wert des Kontostands in die Variable i.
Der Kontostand ist 1, also ist i jetzt gleich 1.
Setzen Sie den Wert des Kontostands auf das Ergebnis von i + 1.
Der Kontostand ist jetzt 2.

❷ Thread B läuft eine Weile

Setzen Sie den Wert des Kontostands in die Variable i.
Der Kontostand ist 2, also ist i jetzt gleich 2.
Setzen Sie den Wert des Kontostands auf das Ergebnis von i + 1.
Der Kontostand ist jetzt 3.
Setzen Sie den Wert des Kontostands in die Variable i.
Der Kontostand ist 3, also ist i jetzt gleich 3.

[Thread B wird jetzt wieder in den Zustand »lauffähig« zurückversetzt, bevor er den Wert des Kontostands auf 4 setzt.]

❸ Thread A läuft weiter und fährt da fort, wo er aufgehört hat

Setzen Sie den Wert des Kontostands in die Variable i.
Der Kontostand ist 3, also ist i jetzt gleich 3.
Setzen Sie den Wert des Kontostands auf das Ergebnis von i + 1,
Der Kontostand ist jetzt 4.
Setzen Sie den Wert des Kontostands in die Variable i.
Der Kontostand ist 4, also ist i jetzt gleich 4.
Setzen Sie den Wert des Kontostands auf das Ergebnis von i + 1.
Der Kontostand ist jetzt 5.

❹ Thread B läuft weiter und fährt genau da fort, wo er aufgehört hat!

Setzen Sie den Wert des Kontostands auf das Ergebnis von i + 1.
Der Kontostand ist jetzt 4. Oje!!

Thread A hat den Kontostand auf 5 erhöht, aber nun kommt Thread B zurück und macht die Aktualisierung von A zunichte, als hätte sie nie stattgefunden.

Wir haben die letzten Aktualisierungen verloren, die Thread A gemacht hat! Thread B hatte vorher schon den Wert des Kontostands »ausgelesen«, und als B aufgewacht ist, hat er einfach weitergemacht, als hätte er nie etwas verpasst.

Machen Sie die Methode inkrementieren() atomar. Synchronisieren Sie sie!

Eine Synchronisierung der Methode inkrementieren() löst das »Problem der verlorenen Aktualisierung«, da sie die beiden Schritte in der Methode zu einer unteilbaren Einheit zusammenschweißt.

```
public synchronized void inkrementieren() {
  int i = kontostand;
  kontostand = i + 1;
}
```

Wenn ein Thread einmal Zutritt zu der Methode erhalten hat, müssen wir dafür sorgen, dass alle Schritte in der Methode zu Ende ausgeführt werden (als ein einziger atomarer Prozess), bevor irgendein anderer Thread in die Methode eintreten kann.

Es gibt keine

Dummen Fragen

F: Klingt, als wäre es eine gute Idee, alles zu synchronisieren - einfach, damit es threadsicher ist.

A: Nein, das ist keine gute Idee. Synchronisierung ist nicht umsonst zu haben. Erst einmal: Eine synchronisierte Methode ist mit einem gewissen Overhead verbunden. Anders ausgedrückt: Wenn Code auf eine synchronisierte Methode trifft, gibt es einen Leistungseinbruch (auch wenn Sie das normalerweise nicht merken würden), während die Frage »Ist der Schlüssel verfügbar?« geklärt wird.

Zweitens kann eine synchronisierte Methode Ihr Programm auch ausbremsen, weil Synchronisierung die Nebenläufigkeit einschränkt. Mit anderen Worten, eine synchronisierte Methode zwingt andere Threads, sich in einer Reihe aufzustellen und zu warten, bis sie dran sind. Vielleicht stellt das in Ihrem Code kein Problem dar, aber Sie müssen es zumindest bedenken.

Und drittens das, was man am meisten fürchten muss: Synchronisierte Methoden können zu einem »Deadlock« führen, d.h., sie können sich gegenseitig blockieren (vgl. Seite 516).

Eine gute Faustregel ist es, sich bei der Synchronisierung auf das notwendige Minimum zu beschränken. Und Sie können tatsächlich sogar winzige Einzelheiten synchronisieren, die noch kleiner sind als eine Methode. Wir brauchen das in diesem Buch zwar nicht, aber Sie können mit dem Schlüsselwort synchronized auch auf der Ebene von ein oder mehreren Anweisungen synchronisieren statt auf der Ebene ganzer Methoden.

tuWas() muss nicht synchronisiert sein, daher synchronisieren wir nicht die gesamte Methode los().

```
public void los() {
  tuWas();

  synchronized(this) {
    tuWasKritisches();
    tuWasNochKritischeres();
  }
}
```

Hier werden nur diese beiden Methodenaufrufe zu einer atomaren Einheit zusammengefasst. Wenn Sie das Schlüsselwort synchronized INNERHALB einer Methode verwenden statt in einer Methodendeklaration, müssen Sie als Argument ein Objekt liefern, dessen Schlüssel sich der Thread holen muss.

Auch wenn es noch andere Möglichkeiten gibt, synchronisiert man doch fast immer auf dem aktuellen Objekt (this). Das ist dasselbe Objekt, dessen Schloss Sie bei einer Synchronisierung der gesamten Methode benutzen würden.

❶ Thread A läuft eine Weile

Versuchen Sie, in die Methode inkrementieren() einzutreten.

Die Methode ist synchronisiert, also **holen Sie sich den Schlüssel** für dieses Objekt.

Setzen Sie den Wert des Kontostands in die Variable i.

Der Kontostand ist 0, also ist i jetzt gleich 0.

Setzen Sie den Wert des Kontostands auf das Ergebnis von i + 1.

Der Kontostand ist jetzt 1.

Geben Sie den Schlüssel zurück (er ist fertig mit der Methode inkrementieren()).

Treten Sie erneut in die Methode inkrementieren() ein und **holen Sie sich den Schlüssel.**

Setzen Sie den Wert des Kontostands in die Variable i.

Der Kontostand ist 1, also ist i jetzt gleich 1,

[Thread A wird jetzt wieder in den Zustand »lauffähig« zurückversetzt, aber weil er die synchronisierte Methode nicht zu Ende ausgeführt hat, behält er den Schlüssel.]

❷ Thread B läuft eine Weile

Versuchen Sie, in die Methode inkrementieren() einzutreten. Die Methode ist synchronisiert, also brauchen wir den Schlüssel.

Der Schlüssel ist nicht verfügbar.

[Thread B wird in einen »Objektschloss-nicht-verfügbar«-Warteraum geschickt.]

❸ Thread A läuft weiter und fährt da fort, wo er aufgehört hat

Setzen Sie den Wert des Kontostands auf das Ergebnis von i + 1.

Der Kontostand ist jetzt 2.

Geben Sie den Schlüssel zurück.

[Thread A wird jetzt wieder in den Zustand »lauffähig« zurückversetzt, aber da er die Methode inkrementieren() beendet hat, behält er den Schlüssel NICHT.]

❹ Thread B läuft weiter und fährt genau da fort, wo er aufgehört hat!

Versuchen Sie, Zutritt zur Methode inkrementieren() zu erhalten. Die Methode ist synchronisiert, also brauchen wir den Schlüssel.

Dieses Mal IST der Schlüssel verfügbar - holen Sie ihn sich!

Setzen Sie den Wert des Kontostands in die Variable i.

[und so weiter und so fort...]