In Java können Sie tatsächlich zur gleichen Zeit laufen und Kaugummi kauen.

Inzwischen dürfte Ihnen klar sein, dass wir uns für Möglichkeit drei entscheiden.

Wir möchten etwas, das kontinuierlich läuft und auf Nachrichten vom Server prüft, *aber ohne dass die Fähigkeit des Benutzers zur Interaktion mit dem GUI beeinträchtigt wird!* Während also der Benutzer fröhlich neue Nachrichten eintippt oder durch die erhaltenen Nachrichten scrollt, möchten wir *hinter den Kulissen* etwas haben, das vom Server eingehende Nachrichten liest.

Das heißt, wir brauchen letztendlich eine Art »Nebenhandlung«: einen neuen Thread. Einen neuen, separaten Stack.

Alles, was wir in der Nur-senden-Version (Version 1) gemacht haben, soll noch wie vorher funktionieren, während nebenher ein neuer *Prozess* ausgeführt wird, der Informationen vom Server liest und sie in dem Textbereich für eingehende Nachrichten anzeigt.

Naja, nicht so ganz. Falls Sie nicht gerade mehrere Prozessoren in Ihrem Rechner haben, ist ein neuer Java-Thread in Wirklichkeit kein separater Prozess auf dem Betriebssystem. Aber man könnte beinahe *glauben*, er wäre es.

Multithreading in Java

Bei Java ist das Multithreading, also die Ausführung mehrerer paralleler »Handlungsstränge«, schon fest in die Sprache eingebaut. Und ein neuer Ausführungsstrang ist fix gemacht:

```
Thread t = new Thread();
t.start();
```

Das war's. Durch die Erzeugung eines neuen Thread-*Objekts* haben Sie einen separaten *Ausführungsstrang – einen Thread –* »vom Stapel gelassen«, mit einem eigenen Aufruf-Stack.

Da wäre aber noch ein Problem.

Dieser Thread tut rein gar nichts und »stirbt« daher im Moment seiner Geburt auch schon seinen »virtuellen Tod«. Wenn ein Thread stirbt, verschwindet auch sein neuer Stack wieder. Ende der Geschichte.

Es fehlt also ein wichtiger Bestandteil - die Aufgabe des Threads. Mit anderen Worten, wir brauchen auch den Code, den ein separater Thread ausführen soll.

In Java bedeutet Multithreading, dass wir uns sowohl den *Thread* als auch die *Aufgabe* ansehen müssen, den er ausführt. Und wir müssen uns die *Klasse* Thread im Paket java.lang ebenfalls ansehen. (Erinnern Sie sich, java.lang ist das Paket, das immer kostenlos und unaufgefordert importiert wird; dort tummeln sich die Klassen, die für die Sprache am grundlegendsten sind, einschließlich String und System.)

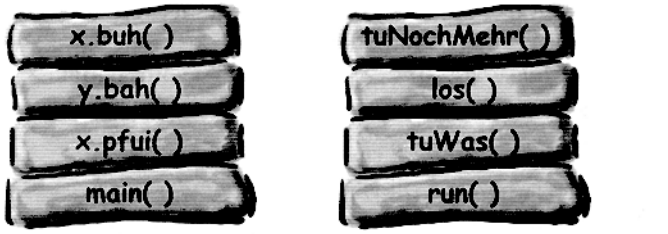
Verwechseln Sie Threads im allgemeinen Sinn nicht mit der Klasse Thread

Wir können über *Threads* im allgemeinen Sinn sprechen und über die Klasse **Thread**. Wenn allgemein von einem *Thread* die Rede ist, ist ein separater Ausführungsstrang gemeint, also ein »Handlungsfaden« mit einem eigenen Geschehen. Mit anderen Worten: ein separater Aufruf-Stack. Es gibt aber auch eine Klasse namens **Thread** im Paket java.lang. Ein **Thread**-Objekt repräsentiert einen Ausführungs-Thread. Jedes Mal, wenn Sie einen neuen *Ausführungs-Thread* starten wollen, werden Sie dafür eine Instanz der Klasse **Thread** erzeugen.

Ein Thread ist ein separater »Ausführungsstrang«, also ein separater Aufruf-Stack.

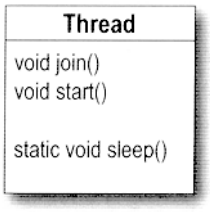
Ein Thread wird in Java von einem Objekt der Klasse Thread repräsentiert.

ein Thread im allgemeinen Sinn



main-Thread — **ein anderer Thread, der vom Code gestartet wurde**

die Klasse Thread



java.lang.Thread

Ein Thread ist ein separater Ausführungsstrang. Ein separater Aufruf-Stack also. Jede Java-Anwendung startet einen main-Thread - den Thread, der main() als unterste Methode auf den Stack setzt. Die JVM ist dafür verantwortlich, den main-Thread zu starten (und weitere Threads, wenn es ihr gefällt, einschließlich des Garbage Collection-Threads). Als Programmierer können Sie Code schreiben, um zusätzlich eigene Threads zu starten.

Thread ist eine Klasse, deren Instanzen einen Ausführungsstrang repräsentieren. Sie enthält Methoden, um einen Thread zu starten, um einen Thread mit einem anderen zu verbinden und um einen Thread »schlafen zu legen«. (Sie enthält noch weitere Methoden; dies hier sind nur die wichtigsten, die wir jetzt brauchen.)

Was heißt es, mehr als einen Aufruf-Stack zu haben?

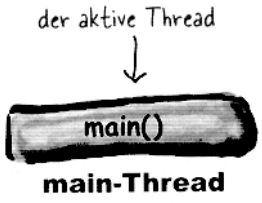
Mit mehr als einem Aufruf-Stack haben Sie den *Eindruck*, dass mehrere Dinge zur gleichen Zeit geschehen. In Wirklichkeit kann nur ein echtes Multiprozessorsystem tatsächlich mehr als eine Sache gleichzeitig tun, aber mit Java-Threads kann es so aussehen, als täte man verschiedene Dinge gleichzeitig. Mit anderen Worten, die Ausführung kann zwischen den Stacks so schnell hin- und herwechseln, dass der Eindruck erweckt wird, alle Stacks würden zur gleichen Zeit ausgeführt. Erinnern Sie sich, Java ist nur ein Prozess, der auf Ihrem darunter liegenden Betriebssystem läuft. Erst einmal muss also Java *selbst* der »aktuell ausführende Prozess« auf dem Betriebssystem sein. Aber wenn Java einmal mit der Ausführung an der Reihe ist, *was* genau führt die JVM dann aus? Welcher Bytecode wird ausgeführt? Die Antwort lautet: das, was gerade oben auf dem aktuell ausgeführten Stack liegt! Und 100 Millisekunden später könnte der aktuell ausgeführte Code zu einer *anderen* Methode auf einem *anderen* Stack hinüberwechseln.

Zu den Aufgaben eines Threads gehört es unter anderem zu verfolgen, welche Anweisung (in welcher Methode) gegenwärtig auf seinem Stack ausgeführt wird.

Das könnte ungefähr so aussehen:

❶ Die JVM ruft main() auf.

```
public static void main(String[] args) {
…
}
```

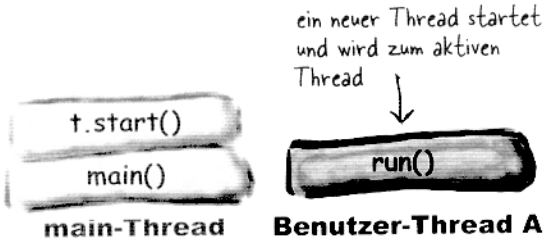


---

❷ main() startet einen neuen Thread. Der main-Thread wird vorübergehend eingefroren während der neue Thread zu laufen beginnt.

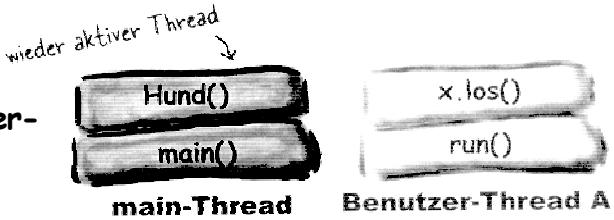
```
Runnable r = new MainThreadJob();
Thread t = new Thread(r);
t.start();
Hund h = new Hund();
```

was das bedeutet, werden Sie gleich sehen …



---

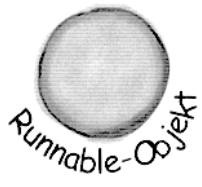
❸ Die JVM wechselt zwischen dem neuen Thread (Benutzer-Thread A) und dem ursprünglichen main-Thread hin und her, bis beide Threads fertig sind.



Wie man einen neuen Thread zur Ausführung bringt:

❶ Ein Runnable-Objekt machen (den Job des Threads)

```
Runnable threadJob = new MeinRunnable();
```



Runnable ist ein Interface, über das Sie auf der nächsten Seite etwas erfahren werden. Sie schreiben eine Klasse, die das Interface Runnable implementiert, und in dieser Klasse legen Sie fest, welche Arbeit der Thread erledigen wird. Mit anderen Worten: welche Methode von dem neuen Aufruf-Stack des Threads ausgeführt wird.

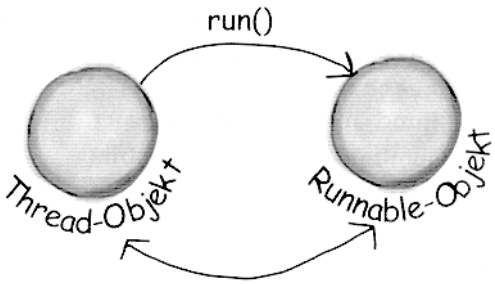
❷ Ein Thread-Objekt (den Arbeiter) machen und ihm ein Runnable (den Job) geben

```
Thread meinThread = new Thread(threadJob);
```

Übergeben Sie dem Thread-Konstruktor das neue Runnable-Objekt. So erfährt das neue Thread-Objekt, welche Methode er als unterste auf seinen neuen Stack setzen soll – die run()-Methode des Runnable-Objekts.

❸ Den Thread starten

```
meinThread.start();
```



Es geschieht erst dann etwas, wenn Sie die Methode start() des Threads aufrufen. Das ist der Punkt, an dem Sie von einer bloßen Thread-Instanz zu einem neuen Ausführungs-Thread übergehen. Wenn der neue Thread startet, nimmt er die run()-Methode des Runnable-Objekts und setzt sie zuunterst auf seinen Stack.

Jeder Thread braucht einen Job. Eine Methode, die auf seinen neuen Stack gesetzt wird.

Runnable ist für einen Thread das, was ein Job für eine Arbeitskraft ist. Ein Runnable ist die Aufgabe, die ein Thread erledigen soll.

Ein Runnable enthält die Methode, die zuunterst auf den Stack des neuen Threads kommt: run().

Ein Thread-Objekt braucht einen Job. Einen Job, den der Thread ausführt, wenn er gestartet wird. Dieser Job ist die erste Methode, die auf den Stack des neuen Threads kommt, und sie muss immer so aussehen:

```
public void run() {
  // Code der vom neuen Thread ausgeführt wird
}
```

Das Interface Runnable definiert nur eine einzige Methode, public void run(). Erinnern Sie sich: Da das ein Interface ist, ist die Methode öffentlich, selbst wenn Sie das nicht extra dazuschreiben.

Woher der Thread weiß, welche Methode er ganz unten auf seinen Stack setzt? Er weiß es, weil das in Runnable vertraglich festgelegt ist. Weil Runnable ein Interface ist. Der Job eines Threads kann in jeder Klasse definiert werden, die das Interface Runnable implementiert. Für den Thread ist nur wichtig, dass Sie seinem Konstruktor ein Objekt einer Klasse übergeben, die Runnable implementiert.

Wenn Sie einem Thread-Konstruktor ein Runnable übergeben, geben Sie dem Thread damit einfach eine Möglichkeit, an eine run()-Methode zu gelangen. Sie geben dem Thread seinen Job.

Um einen Job für Ihren Thread zu erzeugen, implementieren Sie das Interface Runnable

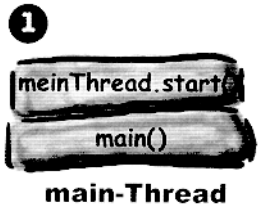
Runnable gehört zum Paket java.lang; sie müssen dieses also nicht importieren.

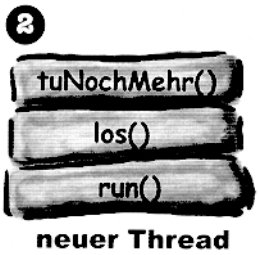
```
public class MeinRunnable implements Runnable {

  public void run() {
    los();
  }

  public void los() {
    tuNochMehr();
  }

  public void tuNochMehr() {
    System.out.println("oben auf dem Stack");
  }
}


class ThreadTestlauf {

  public static void main(String[] args) {

    Runnable threadJob = new MeinRunnable();
    Thread meinThread = new Thread(threadJob);

    meinThread.start();

    System.out.println("zurück in main");
  }
}
```

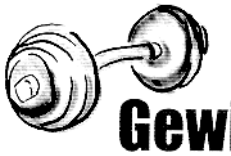
Runnable enthält nur eine einzige zu implementierende Methode: public void run() (ohne Argumente). Hier schreiben Sie den JOB hinein, den der Thread erledigen soll. Dies ist die Methode, die ganz unten auf den neuen Stack kommt.

Übergeben Sie dem neuen Thread-Konstruktor die neue Runnable-Instanz. Damit sagen Sie dem Thread, welche Methode er unten auf den neuen Stack setzen soll. Mit anderen Worten: welche Methode der neue Thread als Erstes ausführen soll.

Sie erhalten erst dann einen neuen Ausführungsstrang, wenn Sie start() auf der Thread-Instanz aufrufen. Ein Thread ist kein richtiger Thread, bevor Sie ihn gestartet haben. Davor ist er einfach eine Instanz der Klasse Thread, wie jedes andere Objekt, aber ihm fehlt die richtige >>Threadhaftigkeit<<.



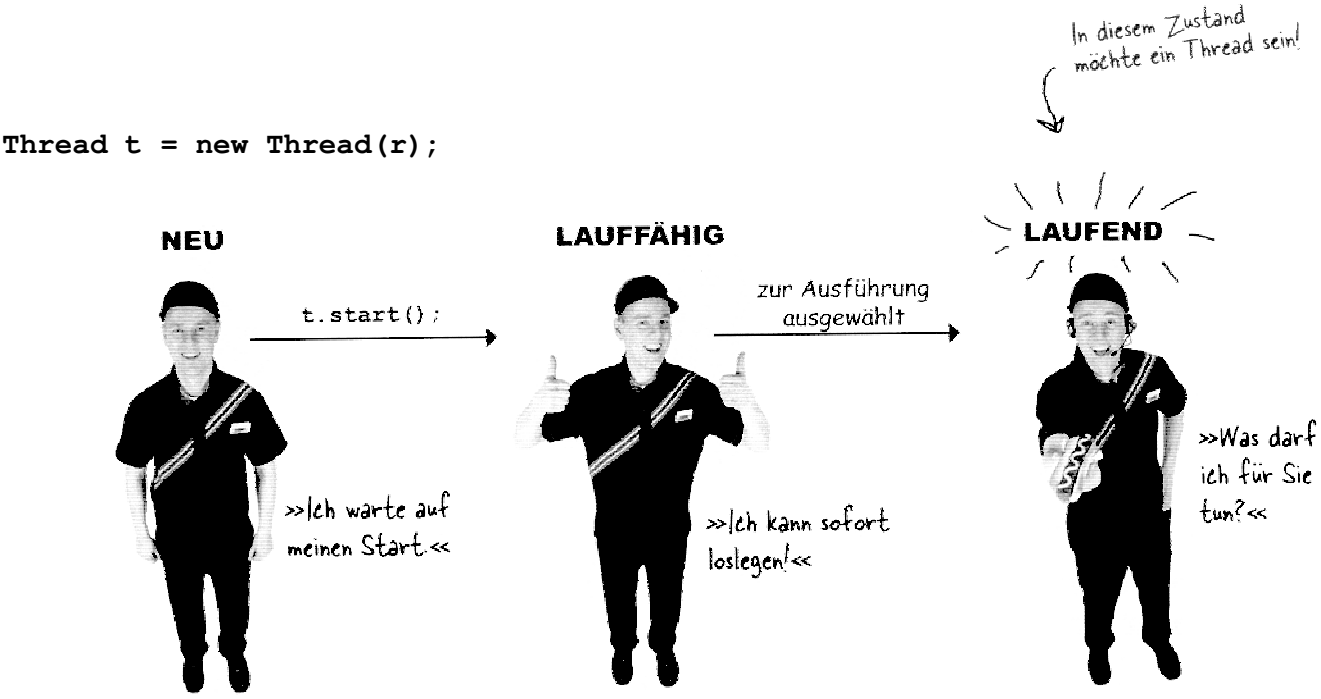




Gewichtheben fürs Gehirn

Wie sieht Ihrer Meinung nach die Ausgabe aus, wenn Sie die ThreadTestlauf-Klasse ausführen? (In ein paar Seiten werden wir das herausfinden.)

Die drei Zustände eines neuen Threads



`Thread t = new Thread(r);`

Eine **neue** Thread-Instanz ist erzeugt, aber noch nicht gestartet worden. Wir haben damit zwar ein Thread-*Objekt*, aber keinen *in Ausführung befindlichen Thread.*

`t.start();`

Wenn Sie den Thread starten, wird er **lauffähig (runnable)**. In diesem Zustand ist er bereit für die Ausführung und wartet nur auf seine große Chance – darauf, dass er zur Ausführung ausgewählt wird. ab jetzt ist für diesen Thread ein neuer Aufruf-Stack vorhanden.

Dies ist der Zustand, nach dem alle Threads lechzen: der *Auserwählte* zu sein! Der jetzt – in diesem Moment – **laufende (running)** Thread. Die Entscheidung liegt ganz allein beim *Thread-Scheduler* der JVM, der für die Ausführungsplanung verantwortlich ist. Sie können seine Entscheidung manchmal *beeinflussen*, aber Sie könnenden Wechsel eines Threads von »lauffähig« zu »laufend« nicht erzwingen. Im Zustand »laufend« besitzt ein Thread (und NUR dieser Thread) einen aktiven Aufruf-Stack, und die Methode oben auf dem Stack wird ausgeführt.

Aber das ist noch nicht alles. Ist der Thread erst einmal lauffähig, kann er zwischen »lauffähig«, »laufend« und einem zusätzlichen Zustand »vorübergehend nicht lauffähig« (auch als »blockiert« bezeichnet) hin und her wechseln.

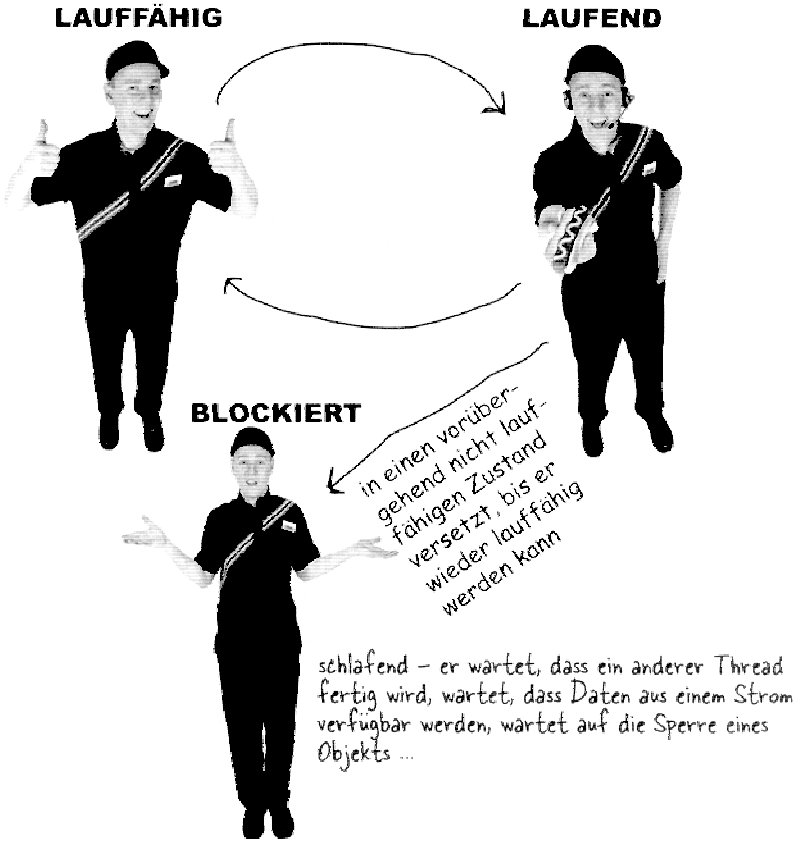
Eine typische lauffähig/laufend-Schleife

Im typischen Fall wechselt ein Thread zwischen lauffähig und laufend hin und her, weil der Thread-Scheduler der JVM einen Thread für die Ausführung auswählt, ihn dann aber wieder zurück »auf die Bank« setzt, um einem anderen Thread die Chance zu geben.

Ein Thread kann »vorübergehend nicht lauffähig« gemacht werden

Der Thread-Scheduler kann einen laufenden Thread aus verschiedenen Gründen in einen **blockierten Zustand** versetzen. Der Thread könnte beispielsweise Code ausführen, der aus einem Socket-Eingabestrom liest – aber es sind gerade keine Daten zum Lesen da. Der Scheduler nimmt dann den Thread aus dem Zustand »laufend« heraus, bis wieder Lesestoff verfügbar ist. Oder vielleicht hat der ausführende Code den Thread angewiesen, sich schlafen zu legen (sleep()). Oder der Thread wartet, weil er versucht, eine Methode auf einem Objekt aufzurufen, das »gesperrt« ist. In diesem Fall kann der Thread erst dann weitermachen, wenn die »Sperre« durch den Thread, der sie gerade benutzt, freigegeben wird.

Unter den genannten (und weiteren) Bedingungen wird ein Thread in den Zustand »vorübergehend nicht lauffähig« versetzt.



Der Thread-Scheduler

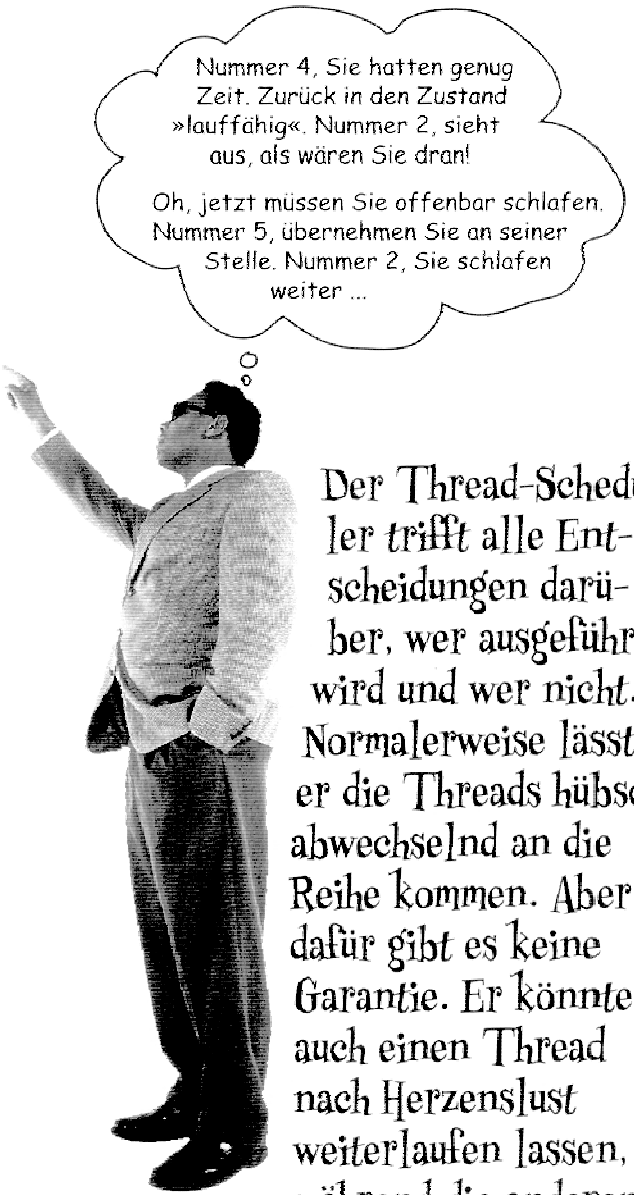
Der Thread-Scheduler ist für die Planung verantwortlich und trifft alle Entscheidungen darüber, welcher Thread vom Zustand »lauffähig« in den Zustand »laufend« versetzt wird und wann (und unter welchen Umständen) ein Thread nicht mehr »laufend« ist. Der Scheduler entscheidet, wer wie lange läuft und was aus den Threads wird, wenn er beschließt, sie aus dem Zustand »laufend« herauszunehmen.

Den Scheduler können Sie nicht steuern. Es gibt keine API für den Aufruf von Methoden auf dem Scheduler. Und vor allem gibt es bei der Planung keinerlei Garantien. (Es gibt ein paar *Fast*-Garantien, aber selbst die sind ein bisschen schwammig.)

Die Schlussfolgerung daraus ist: ***Machen Sie das richtige Funktionieren Ihres Programms nicht davon abhängig, dass der Scheduler in einer bestimmten Art und Weise arbeitet!*** Die Scheduler sind auf verschiedenen JVMs unterschiedlich implementiert, und selbst bei der Ausführung des gleichen Programms auf ein und derselben Maschine können Sie verschiedene Ergebnisse erhalten. Zu den größten Fehlern von Java-Programmieranfängern gehört es, dass sie ihre Multithreading-Programme auf nur einer einzigen Maschine testen und dann annehmen, dass der Thread-Scheduler immer genau auf diese Weise arbeiten wird, egal wo das Programm läuft.

Was bedeutet das also im Hinblick auf die Philosophie »write once, ran anywhere«? Es bedeutet: Wenn Sie plattformunabhängigen Code schreiben wollen, muss Ihr Multithreading-Programm unabhängig vom Verhalten des Thread-Schedulers funktionieren. Sie dürfen sich also beispielsweise nicht darauf verlassen, dass der Scheduler dafür sorgt, dass alle Threads schön fair und gleichberechtigt nacheinander an die Reihe kommen. Auch wenn das heutzutage höchst unwahrscheinlich ist, könnte Ihr Programm doch auf einer JVM mit einem Scheduler landen, der sagt: »Okay, Thread 5, Sie sind dran, und meinetwegen können Sie weitermachen, bis Ihre run()-Methode fertig ist.«

Das Geheimrezept lautet in fast allen Fällen: *schlafen*. Genau, *schlafen*. Wenn Sie einen gerade laufenden Thread schlafen legen, und sei es auch nur für ein paar Millisekunden, zwingen Sie ihn dazu, den Zustand »laufend« zu verlassen, was einem anderen Thread die Chance zur Ausführung gibt. Bei der Thread-Methode sleep() gibt es eine Garantie: dass ein schlafender Thread nicht wieder zum laufenden Thread wird, bevor seine Schlafenszeit rum ist. Wenn Sie beispielsweise Ihren Thread anweisen, zwei Sekunden (2.000 Millisekunden) zu schlafen, kann der Thread auf keinen Fall wieder zum laufenden Thread werden, bevor nicht die zwei Sekunden (oder mehr) verstrichen sind.



Ein Beispiel dafür, wie schlecht sich der Scheduler vorhersagen lässt ...

Die Ausführung dieses Codes auf einer bestimmten Maschine ...

```
public class MeinRunnable implements Runnable {

  public void run() {
    los();
  }

  public void los() {
    tuNochMehr();
  }

  public void tuNochMehr() {
    System.out.println("oben auf dem Stack");
  }
}


class ThreadTestLauf {

  public static void main (String[] args) {

    Runnable threadJob = new MeinRunnable();
    Thread meinThread = new Thread(threadJob);

    meinThread.start();

    System.out.println("zurück in main");
  }
}
```

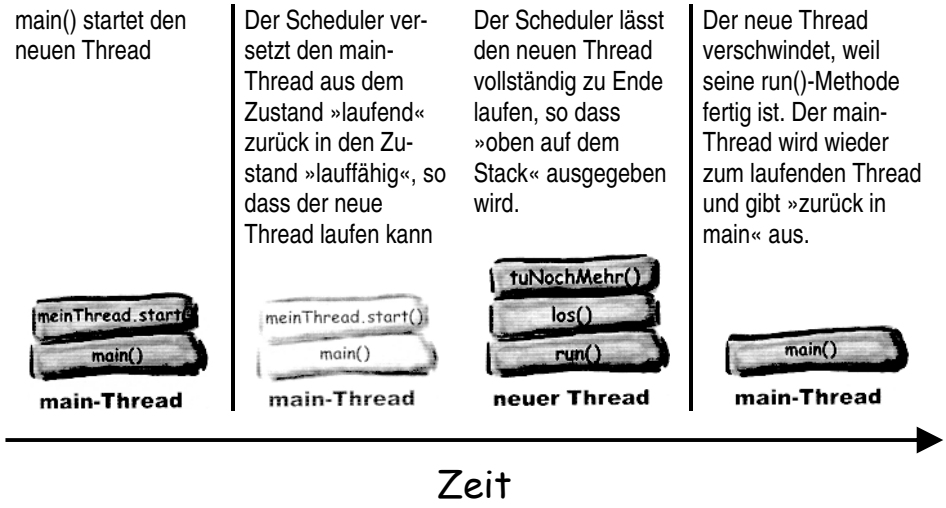
... lieferte diese Ausgabe:

```
Datei  Bearbeiten  Fenster  Hilfe  NimmMichDran
% java ThreadTestlauf
zurück in main
oben auf dem Stack
% java ThreadTestlauf
oben auf dem Stack
zurück in main
% java ThreadTestlauf
oben auf dem Stack
zurück in main
% java ThreadTestlauf
oben auf dem Stack
zurück in main
% java ThreadTestlauf
zurück in main
oben auf dem Stack
% java ThreadTestlauf
oben auf dem Stack
zurück in main
% java ThreadTestlauf
zurück in main
oben auf dem Stack
```

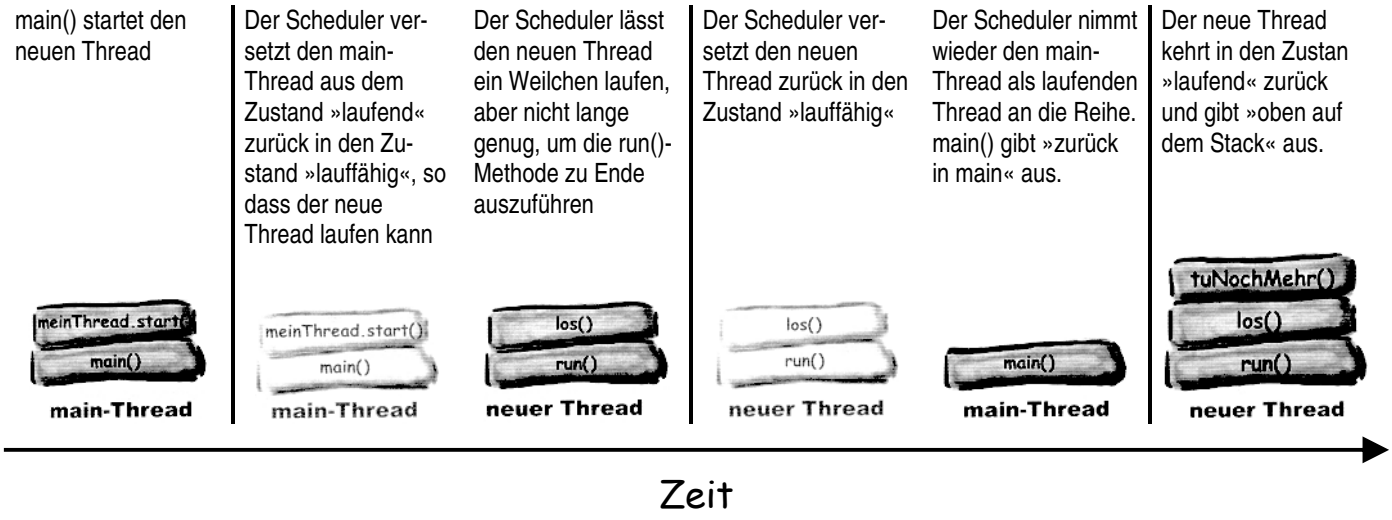
Beachten Sie, dass die Reihenfolge zufällig wechselt. Manchmal ist der neue Thread als Erster fertig und manchmal der main-Thread.

Wie konnten wir unterschiedliche Ergebnisse erhalten?

Manchmal läuft es so:



Und manchmal läuft es so:



Es gibt keine Dummen Fragen

F: Ich habe Beispiele gesehen, die keine separate Runnable-Implementierung verwenden, sondern einfach eine Unterklasse von Thread erzeugen und die Methode run() des Threads überschreiben. Um einen neuen Thread zu erzeugen, ruft man dann den parameterlosen Konstruktor des Threads auf:

```
Thread t = new Thread();   //  kein Runnable
```

A: Ja, so können Sie Ihren Thread tatsächlich auch machen, aber betrachten Sie das einmal aus der OO-Perspektive. Welchen Zweck hat denn die Bildung von Unterklassen? Denken Sie daran, dass wir hier über zwei verschiedene Dinge reden – über den Thread (d.h. das Thread-Objekt) und über den Job des Threads (d.h. des Ausführungsstrangs). Aus OO-Sicht sind dies zwei völlig getrennte Aktivitäten, die in zwei verschiedene Klassen gehören. Eine Unterklasse von Thread sollten Sie nur dann bilden, wenn Sie einen neuen und spezifischeren Typ von Thread machen. Mit anderen Worten: Stellen Sie sich den Thread als Arbeitskraft vor und erweitern Sie die Thread-Klasse nur dann, wenn Sie spezifischeres Verhalten der *Arbeitskraft* benötigen. Aber wenn Sie nichts weiter als einen neuen *Job* brauchen, der von einem Thread/einer Arbeitskraft ausgeführt werden soll, implementieren Sie Runnable in einer separaten, *Job*-spezifischen (nicht Arbeitskraftspezifischen) Klasse.

Dies ist eine Frage des Designs und hat nichts mit der Leistungsfähigkeit oder der Sprache zu tun. Es ist absolut zulässig, Thread zu erweitern und die Methode run() zu überschreiben, aber eine gute Idee ist das nur selten.

F: Können Sie ein Thread-Objekt wiederverwenden? Können Sie ihm einen neuen Job geben und ihn dann durch den Aufruf von start() erneut starten?

Nein. Wenn die run()-Methode eines Threads einmal beendet ist, lässt sich der Thread nie wieder starten. Tatsächlich geht der Thread zu diesem Zeitpunkt in einen Zustand über, den wir noch nicht erwähnt haben – er ist ***tot (dead)***. Im Zustand »tot« hat der Thread seine run()-Methode zu Ende ausgeführt und kann nie wieder gestartet werden. Das Thread-Objekt könnte sich zwar noch auf dem Heap befinden, als »lebendes Objekt«, auf dem Sie andere Methoden aufrufen können (falls das zweckdienlich ist), aber seine »Threadhaftigkeit« hat es für immer verloren. Mit anderen Worten: Es hat keinen separaten Aufruf-Stack mehr und ist damit nicht länger ein Thread im allgemeinen Sinn. Es ist einfach irgendein Objekt - wie alle anderen Objekte.

Es gibt jedoch Entwurfsmuster für die Erzeugung eines Pools von Threads, die man dann zur Ausführung verschiedener Jobs heranziehen kann. Aber das funktioniert nicht so, dass hier ein bereits toter Thread von Neuem gestartet würde.

Punkt für Punkt

- Ein Thread im allgemeinen Sinn ist in Java ein separater Ausführungsstrang.
- Jeder Thread hat in Java seinen eigenen Aufruf-Stack.
- Ein Thread-Objekt – eine Instanz der Klasse java.lang.Thread – repräsentiert einen Ausführungsstrang.
- Ein Thread braucht einen Job. Der Job eines Threads ist eine Instanz von etwas, das das Interface Runnable implementiert.
- Das Interface Runnable enthält nur eine einzige Methode, run(). Dies ist die Methode, die an unterster Stelle des neuen Aufruf-Stacks steht. Mit anderen Worten: Es ist die erste Methode, die in dem neuen Thread ausgeführt wird.
- Um einen neuen Thread in Gang zu setzen, brauchen Sie ein Runnable, das Sie dem Konstruktor des Threads übergeben.
- Ein Thread ist im Zustand NEU (NEW), wenn Sie ein Thread-Objekt instantiiert, aber noch nicht start() aufgerufen haben.
- Wenn Sie einen Thread starten (durch Aufrufen der Methode start() des Thread-Objekts), wird ein neuer Stack erzeugt, an dessen unterster Stelle die run()-Methode des Runnable-Objekts steht. Der Thread ist jetzt im Zustand LAUFFÄHIG (RUNNABLE) und wartet darauf, für die Ausführung ausgewählt zu werden.
- Ein Thread wird als LAUFEND (RUNNING) bezeichnet, wenn der JVM-Thread-Scheduler ihn als laufenden Thread ausgewählt hat. Auf einer Einprozessormaschine kann es immer nur einen gerade laufenden Thread geben.
- Manchmal kann ein Thread vom Zustand LAUFEND in den Zustand BLOCKIERT (BLOCKED) versetzt werden, d.h., er ist vorübergehend nicht lauffähig. Ein Thread kann z.B. blockiert sein, weil er auf Daten aus einem Strom wartet oder weil er schlafen gegangen ist oder weil er auf die Sperre eines Objekts wartet.
- Es wird nicht garantiert, dass der Thread-Scheduler auf eine bestimmte Art und Weise arbeitet; daher können Sie nicht sicher sein, dass Threads sich abwechseln. Sie können dies jedoch unterstützen, indem Sie Ihre Threads regelmäßig schlafen schicken.

Einen Thread schlafen legen

Eine der besten Möglichkeiten, Threads abwechselnd drankommen zu lassen, ist, sie regelmäßig schlafen zu schicken. Dazu müssen Sie nur die statische Methode sleep() aufrufen und Ihr die Schlafdauer (in Millisekunden) übergeben.

Ein Beispiel:

```
Thread.sleep(2000);
```

Dies nimmt einen Thread aus dem Zustand »laufend« heraus und hält ihn zwei Sekunden lang davon ab, wieder in den Zustand »lauffähig« zurückzukehren. Er kann *nicht* wieder zum laufenden Thread werden, bevor nicht mindestens zwei Sekunden vergangen sind.

Etwas bedauerlich ist, dass die sleep-Methode eine InterruptedException auslöst, eine geprüfte Exception. Alle Aufrufe von sleep müssen daher mit einem try/catch-Block umgeben werden (oder die Exception muss deklariert werden). Ein Aufruf von sleep sieht daher in der Realität so aus:

```
try {
  Thread.sleep(2000);
} catch (InterruptedException ex) {
  ex.printStackTrace();
}
```

Es ist allerdings unwahrscheinlich, dass jemals etwas Ihren Thread beim Schlafen unterbrechen wird; die Exception steht in der API, um einen Thread-Kommunikationsmechanismus zu unterstützen, den im wirklichen Leben fast niemand verwendet. Trotzdem müssen Sie sich an das Gesetz vom Behandeln oder Deklarieren halten – gewöhnen Sie sich einfach daran, Ihre sleep()-Aufrufe mit try/catch zu umgeben.

Sie wissen nun, dass Ihr Thread nicht aufwachen wird, *bevor* die angegebene Zeit verstrichen ist, aber ist es möglich, dass er erst eine Weile *nach* dem Ablaufen des »Weckers« aufwacht? Die Antwort ist: ja und nein. Es ist eigentlich auch egal, denn wenn der Thread aufwacht, ***geht er immer automatisch in den Zustand »lauffähig«!*** Er wacht nicht automatisch zur festgesetzten Zeit auf und wird sofort zum laufenden Thread. Wenn ein Thread aufwacht, ist er wieder einmal der Gnade des Thread-Schedulers ausgeliefert. Bei Programmen, die kein perfektes Timing erfordern und nur wenige Threads haben, kann es zwar so aussehen, als ob der Thread aufwacht und genau nach Plan direkt weiter ausgeführt wird (z.B. nach 2.000 Millisekunden). Aber verwetten Sie nicht Ihr Programm darauf!

Legen Sie Ihren Thread schlafen, wenn Sie sicher sein wollen, dass auch andere Threads eine Chance zum Laufen erhalten.

Wenn der Thread aufwacht, befindet er sich immer zunächst im Zustand »lauffähig« und wartet darauf, dass ihn der Thread-Scheduler wieder für die Ausführung aussucht.

Erzeugen und Starten von zwei Threads

Threads haben Namen. Sie können Ihre Threads auf einen Namen Ihrer Wahl taufen oder ihre vorgegebenen Namen akzeptieren. Das Tolle an den Namen ist, dass Sie damit feststellen können, welcher Thread gerade läuft. Das folgende Beispiel startet zwei Threads. Jeder Thread hat die gleiche Aufgabe: in einer Schleife zu laufen und bei jeder Iteration den Namen des aktuell laufenden Threads auszugeben.

```
public class ZweiThreads implements Runnable {

  public static void main(String[] args) {
    ZweiThreads aufgabe = new ZweiThreads();
    Thread alpha = new Thread(aufgabe);
    Thread beta = new Thread(aufgabe);
    alpha.setName("Alpha-Thread");
    beta.setName("Beta-Thread");
    alpha.start();
    beta.start();
  }
  public void run() {
    for (int i = 0; i < 25; i++) {
      String threadName = Thread.currentThread().getName();
      System.out.println("jetzt läuft der " + threadName);
    }
  }
}
```

Machen Sie eine Instanz von Runnable.

Machen Sie zwei Threads mit dem gleichen Runnable (also dem gleichen Job – in ein paar Seiten sagen wir noch mehr zum Thema >>zwei Threads mit nur einem Runnable<<).

Geben Sie den Threads Namen.

Starten Sie die Threads.

Jeder Thread durchläuft diese Schleife und gibt bei jedem Durchgang seinen Namen aus.

Was wird passieren?

Teil der Ausgabe bei 25facher Wiederholung der Schleife.

```
Datei  Bearbeiten  Fenster  Hilfe  Centauri
jetzt läuft der Alpha-Thread
jetzt läuft der Alpha-Thread
jetzt läuft der Alpha-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Alpha-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Alpha-Thread
```

Werden die Threads sich abwechseln? Werden Sie mal diesen, mal jenen Thread-Namen sehen? Wie oft wird es wechseln? Bei jedem Schleifendurchlauf? Alle fünf Durchläufe?

Sie kennen die Antwort bereits: Wir wissen es nicht! Es hängt vom Scheduler ab. Und auf Ihrem Betriebssystem, mit Ihrer speziellen JVM, auf Ihrer CPU erhalten Sie vielleicht ein ganz anderes Ergebnis als wir.

Mit OS X 10.2 (Jaguar) wird bei fünf oder weniger Iterationen erst der Alpha-Thread zu Ende ausgeführt und dann der Beta-Thread. Sehr einheitlich. Nicht 100%ig, aber sehr einheitlich.

Wenn Sie jedoch die Schleife auf 25 oder mehr Durchläufe setzen, kommen die Dinge ins Wackeln. Dann schafft der Alpha-Thread vielleicht nicht alle 25 Iterationen, bevor ihn der Scheduler wieder in den Zustand »lauffähig« zurückschiebt, um dem Beta-Thread eine Chance zu geben.