Typische Anwendungen sind einmalige Ereignisse (in Abschnitt 2.3.3 zum Stoppen eines Threads) oder die Veröffentlichung von Informationen (zum Beispiel Temperaturmessungen), die zustandslos sind.

Wie oben erwähnt wurde, entspricht der Zugriff auf eine `volatile`-Variable einem Punkt, bei dem der Cache mit dem Hauptspeicher synchronisiert wird. Dies ist eine relativ »teure« Angelegenheit. Zur Veröffentlichung von Aktualisierungen sind andere Lösungen (z.B. über das *Observer*-Muster) unter Umständen effizienter.

3.5 Final-Attribute

Neben den Sichtbarkeitsregeln für `volatile` gibt es auch Entsprechendes für mit `final` gekennzeichnete Attribute. Es gilt, dass sie entweder direkt bei der Deklaration oder im Konstruktor initialisiert werden können. Alle `final`-Attribute sind nach der Objekterzeugung komplett initialisiert und nicht mehr veränderbar. Dabei bezieht sich die »Unveränderbarkeit« nur auf das Attribut selbst. Handelt es sich bei dem Attribut um eine Referenz, so wird das `final` nicht auf den »Inhalt« des referenzierten Objekts ausgedehnt.

Sichtbarkeitsregeln für referenzierte Objekte

Werden über `final`-Attribute Objekte referenziert, so kann es beim falschen Gebrauch in einer Multithreaded-Umgebung zu unerwünschten Effekten kommen. Liest ein Thread zum ersten Mal ein `final`-Attribut, so werden auch alle über die Referenz erreichbaren Objekte, die sogenannte *transitive Hülle* gelesen und dauerhaft in den Cache übertragen. Es ist wichtig zu wissen, dass sich die transitive Hülle in die Tiefe über mehrere Ebenen des Objektgraphen erstrecken kann.

Solange alle beteiligten Objekte *immutable* sind, kann es zu keinem Seiteneffekt bei nebenläufigen Zugriffen kommen[5]. Ist das nicht der Fall, dann muss man aufpassen, da Veränderungen nicht unbedingt für andere Threads gleich sichtbar sind. Es sei denn, man benutzt hier die Mechanismen `volatile` oder `synchronized`[6].

[5] In dem Fall kann sich auch die Gestalt der transitiven Hülle nicht ändern.

[6] Das Schlüsselwort `synchronized` wird in Kapitel 4 besprochen.

3.6 Thread-lokale Daten

Wie wir oben gesehen haben, kann ein Thread während der Ausführung auf verschiedene lokale und globale Daten zugreifen. Viele Betriebssysteme unterstützen darüber hinaus einen sogenannten *Thread-lokalen Speicher* (*thread-specific* oder *thread-local storage* oder kurz *TLS* genannt). Technisch gesehen ist der TLS ein spezieller Speicherbereich, der nur einem Thread gehört und dessen Zeiger im Thread-Kontext abgespeichert wird. Bei jedem Kontextwechsel wird damit der Speicherbereich des aktuellen Threads geladen. Objekte, die im TLS abgelegt werden, sind im folgenden Sinne *quasi-global*:

- Sie sind global, weil die Namen der Objekte global (Klassenvariablen) bekannt sind.
- Sie sind aber in dem Sinne lokal, weil mit jedem Thread-Wechsel das Objekt ausgetauscht wird, obwohl es einen globalen Namen hat. Jeder Thread hat ein eigenes zugehöriges Objekt.

Objekte im TLS werden in Java durch die Klasse `ThreadLocal<T>` unterstützt. Um die Plattformunabhängigkeit zu gewährleisten, wird eine globale Datenstruktur verwendet. Die Klasse `ThreadLocal` besitzt neben dem Defaultkonstruktor die Methode `set(T value)`, mit der ein Objekt vom Typ `T` in den Thread-lokalen Speicher abgelegt werden kann. Mit `remove` kann es gelöscht werden. Durch Überschreibung der `initialValue`-Methode kann man dem Objekt einen Defaultwert zuweisen. Seit Java 8 kann alternativ die Fabrikmethode `ThreadLocal.withInitial(Supplier<T>)` benutzt werden.

Eine typische Anwendung von `ThreadLocal`s ist die Realisierung eines »Per-Thread-*Singleton*«. Bei JEE-Servern wird dies z. B. für das Ablegen von Kontextinformationen benutzt (z. B. Transaktions-ID). Ein weiteres Beispiel ist `ThreadLocalRandom` (siehe unten).

Praxistipp

Namen der Thread-lokalen Objekte sind typischerweise global und werden daher `static` deklariert, um die gewünschte Wirkung zu erzielen. Dadurch wird sichergestellt, dass alle Threads auf `ThreadLocal`-Objekte über denselben Namen zugreifen können (vgl. [34, 16, 45]). Manchmal sieht man, dass sie in gewöhnlichen Instanzattributen (ohne `static`) hinterlegt werden, was leider unter Umständen nicht die gewünschte Wirkung hat.

ThreadLocalRandom

Um den Umgang mit Thread-lokalen Objekten zu verdeutlichen, betrachten wir eine Situation, in der verschiedene Threads sehr viele Zufallszahlen benötigen (etwa bei einer Monte-Carlo-Simulation). Die naive Lösung wäre die Verwendung der Klassenmethode `Math.random`, um sie zu erzeugen. Dies führt zu einem problematischen Engpass. Der verwendete Generator besitzt einen »Zustand«. Die Aufrufe müssen Thread-sicher gestaltet werden (z.B. mit dem Atomic-Konzept – siehe Kapitel 7). Das verursacht natürlich einen Engpass beim Durchsatz. Eine Lösungsidee ist, dass jeder Thread seinen eigenen Zufallsgenerator (z.B. ein Objekt der Klasse `java.util.Random`) benutzt. Hierzu gibt es verschiedene Implementierungsmöglichkeiten:

1. Jedes Thread- bzw. `Runnable`-Objekt besitzt ein Attribut für einen eigenen Zufallsgenerator. In der `run`-Methode wird dann darauf zugegriffen.
2. Mithilfe des Thread-lokalen Konzepts kann jedem Thread einheitlich ein Zufallsgenerator zur Verfügung gestellt werden.
3. Man benutzt die von Java zur Verfügung gestellte `ThreadLocalRandom`-Klasse, die einer Implementierung der vorhergehenden Idee entspricht.

Codebeispiel 3.1 zeigt die Verwendung Thread-lokaler Daten.

```
public class MyThreadLocalRandom
{
  static ThreadLocal<Random> rand = new ThreadLocal<Random>()     ❶
  {
    @Override
    protected Random initialValue()                               ❷
    {
      return new Random();
    }
  };

  public static void main(String[] args)
  {
    for (int i = 0; i < 10; i++)     // erzeuge 10 Threads
    {
      new Thread(()->{
        StringBuffer strBuf = new StringBuffer();
        strBuf.append(Thread.currentThread().getId() + " : ");
        for (int j = 0; j < 100; j++)
           strBuf.append( rand.get().nextInt(100) + " ");          ❸
        System.out.println(strBuf);
    }).start();
   }
  }
}
```

Codebeispiel 3.1: Beispiel für eine Anwendung Thread-lokaler Daten

Dem statischen Attribut `rand` wird ein Objekt einer Subklasse von `ThreadLocal<Random>` zugewiesen (❶), wobei hier die Methode `initialValue` überschrieben wird (❷). Eine Alternative ab Java 8 wäre die Verwendung von `ThreadLocal.withInitial( Random::new )`. Wird von einem Thread das erste Mal die `get`-Methode aufgerufen (❸), wird `initialValue` ausgeführt und ein `Random`-Objekt in den Thread-lokalen Speicher abgelegt. Der Aufruf `rand.get` liefert dann im weiteren Verlauf immer das vorher zugewiesene Objekt.

Die Lösung im Codebeispiel 3.1 ist leider ineffizient. Der Grund ist die (schlechte) Implementierung der Klasse `Random`. Die Methoden von `Random` wurden Thread-sicher gestaltet, was bei der obigen Anwendung einen unnötigen Overhead produziert. Seit Java 7 gibt es für diesen Einsatz als Alternative die `ThreadLocalRandom`-Klasse, die von `Random` erbt und alle Objektmethoden so überschreibt, dass die Thread-Sicherheit nicht mehr benötigt wird. Mit der Klassenmethode `ThreadLocalRandom.current` kann auf den entsprechenden Generator zugegriffen werden. Das obige Beispiel vereinfacht sich dann zu:

```
public class MyThreadLocalRandom
{
  public static void main(String[] args)
  {
    // erzeuge 5 Threads
    for (int i = 0; i < 5; i++)
    {
      new Thread(() ->
       {
         StringBuffer strBuf = new StringBuffer();
         strBuf.append(Thread.currentThread().getId() + " : ");
         for (int j = 0; j < 10; j++)
         {
           // verwende Thread-eigenen Zufallsgenerator
           strBuf.append(
               ThreadLocalRandom.current().nextInt(100) + " ");
         }
         System.out.println(strBuf);
       }).start();
    }
  }
}
```

Codebeispiel 3.2: Anwendung von ThreadLocalRandom

3.7 Fallstricke

- **Modifikationen von volatile-Variablen sind nicht atomar**
  Durch die Spezifikation wird lediglich zugesichert, dass das Lesen und Schreiben von `volatile`-Variablen atomar sind. Das Codebeispiel 3.3 arbeitet nicht korrekt.

```
class Counter
{
  private volatile int count = 0;

  public Counter() { super(); }
  public int getNext() { return ++count; }
}
```

Codebeispiel 3.3: Inkorrekte Implementierung eines Counters

  Die Variable `count` ist hier zwar mit `volatile` gekennzeichnet, die Anweisung `++count` ist allerdings nicht atomar. Beim gleichzeitigen Zugriff auf `getNext` kann es durchaus vorkommen, dass zweimal derselbe Wert zurückgeliefert wird.

  Abhilfe schafft hier die Kennzeichnung von `getNext` mit `synchronized` (dann kann in dem Beispiel auf `volatile` verzichtet werden) oder die Benutzung eines `AtomicInteger`-Objekts (siehe Kapitel 7).

- **Ausgelassener volatile-Zugriff**
  Das Codebeispiel 3.4 enthält eine subtile Schwachstelle. Hier führt die Wirkung der *Shortcut*-Auswertung zu einem Problem.

```
boolean cond1 = true;
volatile boolean cond2 = true;
long counter = 0;

@Override
public void run()
{
  while ( cond1 || cond2 )
  {
    counter++;
  }
  System.out.println("worker done : " + counter);
}
```

Codebeispiel 3.4: Auslassen eines volatile-Zugriffs

  Wenn `cond1` `true` ist, kommt die Speicherbarriere von `cond2` nicht zur Geltung.

■ **Falsche Verwendung von `ThreadLocalRandom`**
Im folgenden Codebeispiel wird `ThreadLocalRandom` falsch verwendet.

```
public static Random rand = ThreadLocalRandom.current();
```

Codebeispiel 3.5: Falsche Verwendung von ThreadLocalRandom

Jeder später erzeugte Thread, der direkt `rand` benutzt, generiert hier dieselbe Zufallsfolge. Der Grund ist die nicht konforme Implementierung des `ThreadLocal`-Konzepts. `ThreadLocalRandom.current` muss zwingend im Thread-Kontext aufgerufen werden, damit der eigene Zufallsgenerator korrekt initialisiert wird. In dem Beispiel wird `ThreadLocalRandom.current` nur im Kontext des *Classloaders* aufgerufen.

3.8 Zusammenfassung

Unveränderbare (*immutable*) Objekte können problemlos von mehreren Threads gelesen werden. Sind die Daten bzw. Objekte veränderbar, müssen geeignete Synchronisationsmechanismen verwendet werden. Je nach Aufgabe kann der Einsatz von `volatile` ausreichend sein.

Wird eine `volatile`-Variable modifiziert, so werden alle davor gemachten Änderungen aus dem Cache des jeweiligen Threads in den Hauptspeicher übertragen (*flush*). Beim Lesen von `volatile`-Attributen wird der Cache aktualisiert (*refresh*). Greifen zwei Threads auf verschiedene `volatile`-Variablen zu, so existieren für sie keine Sichtbarkeitsgarantien für gemeinsam benutzte non-`volatile` Daten.