Übungen zu Kapitel 21

21.1 Namen von Threads: Neben den bereits bekannten Konstruktoren existieren zusätzliche Konstruktoren

```
Thread( String name )
Thread( Runnable target, String Name )
```

mit denen der Name des neuen Threads gesetzt werden kann. Ändern Sie das Beispiel `CountThread`, indem Sie beim Erzeugen der Threads einen Namen mitgeben. Da Konstruktoren nicht vererbt werden, ist der Konstruktor `MyThread(String name)` ebenfalls zu implementieren. Formulieren Sie das Beispiel auch in der Runnable-Variante unter Verwendung des zweiten Konstruktors.

21.2 Neue Threads können nicht nur aus dem main-Thread sondern auch aus erzeugten Threads gestartet werden. Leiten Sie die Klasse `ManyThreads` von `Thread` ab und überschreiben Sie die Methode `run()`, so dass dort ein Exemplar von `ManyThreads` erzeugt und gestartet sowie der Name des aktuell laufenden Threads am Bildschirm ausgegeben wird. Starten Sie in der `main()`-Methode ein Exemplar von `ManyThreads` und beobachten Sie den Output des laufenden Programms.

Wundern Sie sich nicht, wenn das Programm immer neue Threads erzeugt und nicht terminiert. Erweitern Sie das Programm, so dass mit dem Aufruf `new ManyThreads( howmany ).start();` insgesamt genau `howmany` viele Threads gestartet werden. Erweitern Sie die Klasse um ein Attribut `private int howmany` und einen entsprechenden Konstruktor. Der Konstruktoraufruf in `run()` erfolgt dann - falls überhaupt noch Threads zu erzeugen sind - mit einem um 1 verminderten Wert.

21.3 Starten Sie in einem Testprogramm mehrere Threads und versuchen Sie, einen mehrmals zu starten. Was passiert? Welche Threads sind betroffen?

21.4 Erstellen Sie ein Testprogramm, das den Namen des main-Threads ausgibt.
Tipp: Beschaffen Sie sich mit `currentThread()` eine Referenz auf das aktuelle Thread-Objekt.

21.5 Starten im Konstruktor: Anstatt ein Thread-Objekt zu erzeugen und dessen Methode `start()` anzuwenden, kann `start()` auch direkt im Konstruktor platziert werden, so dass der Thread beim Erzeugen von alleine losläuft. Ergänzen Sie die Klasse `MyThread` um den Konstruktor

```
public MyThread(){   start();   }
```

und entfernen Sie im ausführbaren Programm `ErstesThreadProgramm` den Aufruf von `start()`. Verifizieren Sie, dass das Programm wie gewohnt abläuft.

Dieses Vorgehen hat leider auch einige Nachteile, insbesondere dann, wenn man von einer solchen "sich selbst startenden" Klasse ableitet. Werden nämlich Objekte der abgeleiteten Klasse erzeugt, wird implizit der Konstruktor der Oberklasse ausgeführt, so dass der Thread bereits losläuft, bevor das Objekt im eigentlichen Konstruktor fertig gestellt ist. Bestätigen Sie diese Bedenken anhand folgender Klassen:

```
public class ThreadA extends Thread {
   public int me;
   public ThreadA() {  start(); }
}
public class ThreadB extends ThreadA {
   public ThreadB( int me ){
      yield();
      this.me = me;
   }
   public void run() {
      System.out.println( me );
      yield();
      System.out.println( me );
   }
}
```

Erzeugen Sie in einem ausführbaren Programm eine Instanz von `ThreadB` und erklären Sie die Ausgabe. Die Methode `yield()` wird hierbei verwendet, um der JVM das Umschalten auf andere Threads zu ermöglichen.

21.6 Formulieren Sie analog zur vorherigen Aufgabe in der Klasse `MyRunnable` einen Konstruktor, der den gewünschten Thread direkt beim Erzeugen eines `MyRunnable`-Objektes startet.

21.7 start() und run(): Falls man aus Versehen für ein Thread-Objekt anstelle von `start()` die Methode `run()` aufruft, wird diese Methode direkt im ursprünglichen Thread ausgeführt. Erstellen Sie ein Testprogramm mit folgender `main()`-Methode:

```
public static void main( String[] args ){
   new CountThread().run();
   new CountThread().run();
}
```

Wie man am Output des ausgeführten Programmes feststellt, wird der Name des Thread-Objektes ausgegeben, nicht aber der Name des ausführenden Threads. Erweitern Sie die Methode `run()`, so dass neben dem Namen des Thread-Objektes auch der Name des ausführenden Threads (`currentThread().getName()`) angezeigt wird und führen Sie das Programm erneut aus.

Erweitern Sie die `run()`-Methode, so dass diese stets im "eigenen Thread" ausgeführt wird. Tipp: Falls Thread-Objekt und `currentThread()` nicht übereinstimmen, löst man eine nicht-kontrollierte Ausnahme (zB `RuntimeException`) aus.

21.8 Performance von synchronized-Methoden: Vergleichen Sie für das PlusPlus- und das Tankprogramm jeweils die Laufzeiten der korrekten (`synchronized`) Version mit der inkorrekten Version. Das Ergebnis zeigt, dass die Sperrverwaltung in Java sehr ressourcenintensiv ist und auch wirklich nur dann eingesetzt werden sollte, wenn dies notwendig ist.

21.9 Runnable: Formulieren Sie das Zeitanzeigeprogramm in der Runnable-Variante, indem Sie die Klassen `ZeitanzeigeRunner` und `ZeitanzeigeRunnerProgramm` erstellen. Die erste Klasse soll `Runnable` implementieren, wobei die `run()`-Methode fast identisch aus der Klasse `Zeitanzeige` übernommen werden kann; die zweite Klasse erzeugt in der `main()`-Methode ein `ZeitanzeigeRunner`-Objekt und mit diesem und dem entsprechenden Konstruktor `Thread(...)` ein `Thread`-Objekt, das anschließend gestartet wird. Beachten Sie, dass für das Attribut `stopped` eine Referenz auf das `Runnable`-Objekt benötigt wird.

21.10 Erweitern Sie im Tankprogramm die Klasse `Pumpe` um die öffentliche Methode `getSum()` und geben Sie am Ende der `main()`-Methode aus, ob `p1.getSum() + p2.getSum()` mit `t.getInhalt()` übereinstimmt oder nicht.

Erweitern Sie in einem zweiten Schritt die Klasse Tankprogramm um eine Konstante `static final int LEN = 10` und legen Sie einen Array `p` von `LEN`-vielen Pumpen an:

```
Pumpe[] p = new Pumpe[10];
for( int i=0; i<LEN; i++ )
   p[i] = new Pumpe( "Pumpe-" + ... );
```

Anschließend soll - analog zur ursprünglichen Version des Programms - jede Pumpe als Thread gestartet, auf das Ende aller Threads gewartet und am Ende die Summe der von den Pumpen betankten Mengeneinheiten mit dem Tankinhalt verglichen werden. Testen Sie jeweils mit und ohne Schlüsselwort **`synchronized`**.

21.11 Lost Update: Das Lieferantenprogramm ist noch nicht korrekt formuliert, da ein ***lost-update*** Problem auftreten kann. In der vorliegenden Version ist es jedoch sehr unwahrscheinlich, dass dieses Problem zu Tage tritt, da jeder Thread die Methode `anliefern()` nur einmal ausführt. Um die Synchronisationsprobleme zu provozieren, ändern wir die Methode `run()` in der Klasse `Lieferant` wie folgt ab:

```
public void run() {
   for( int i = 0; i < 1000; i++ ) {
      anliefern( 1 );
   }
}
```

Führen Sie diese Änderungen durch und verifizieren Sie, dass der Endbestand nicht mit der Summe der von den Lieferanten gelieferten Mengeneinheiten übereinstimmt.

Um das Problem zu lösen, genügt es jedoch ***nicht***, die Methode `anliefern()` der Klasse `Lieferant` mit `synchronized` zu kennzeichnen. Das zugehörige Sperrobjekt dieser Instanzmethode wäre nämlich der jeweilige Lieferant, so dass verschiedene Lieferanten durchaus gleichzeitig anliefern könnten (die Sperre bezieht sich ja auf unterschiedliche Objekte).

Als Sperrobjekt kommt letztlich nur das Lager in Frage, so dass eine mögliche Lösung darin besteht, eine Instanzmethode `einlagern()` der Klasse `Lager` (als Pendant zur Methode `anliefern()`) zu verwenden und diese mit `synchronized` zu kennzeichnen. Erweitern Sie die Klassen wie folgt und führen Sie das Programm aus:

```
// in Lieferant
public void run() {
   for( int i = 0; i < 1000; i++ ) {
      // anliefern( 1 );
      l.einlagern( 1 );
   }
}

// in Lager
public synchronized void einlagern( int anzahl ){
   int bestand = getBestand();
   System.out.print( "Bestand alt: " + bestand );
   System.out.println( "Liefermenge: " + anzahl );
   setBestand( bestand + anzahl );
}
```

Da nun unabhängig von den beteiligten Lieferanten das gleiche Sperrobjekt, nämlich das Lager `l`, verwendet wird, können keine zwei Lieferanten die Methode `einlagern()` verschränkt ausführen.

21.12 Synchronized Blöcke: Eine alternative Lösungsmöglichkeit für das Lieferantenprogramm besteht darin, `synchronized`-***Blöcke*** anstelle von `synchronized`-***Methoden*** zu verwenden. Grundsätzlich können nicht nur ganze Methoden sondern auch einzelne Blöcke mit `synchronized` gekennzeichnet werden, wobei das Sperrobjekt zusätzlich angegeben werden muss. Dieses Verfahren bietet mehr Flexibilität, da beliebige Objekte als Sperrobjekte verwendet werden können. Bei `synchronized`-Methoden ist das Sperrobjekt bereits festgelegt, nämlich das aufrufende Objekt bei einer Instanzmethode bzw. die Klasse bei einer statischen Methode.

Modifizieren Sie das ursprüngliche Programm, indem Sie den Rumpf der Methode `anliefern()` als `synchronized`-Block mit dem Lager als Sperrobjekt folgendermaßen kennzeichnen:

```
public void anliefern(int anzahl) {
   synchronized( l ) {
      int bestand = l.getBestand();
      System.out.print( getName() + " Bestand alt: " + bestand );
      System.out.println( "  Liefermenge: " + anzahl );
      l.setBestand( bestand + anzahl );
   }
}
```

Alternativ kann auch der Aufruf von `anliefern()` in der Methode `run()` in einen entsprechenden `synchronized`-Block gepackt werden. Probieren Sie dies aus.

21.13 Sichtbarkeit von Werten und das Schlüsselwort `volatile`

Im Zeitanzeigeprogramm des Kapitels wird das Attribut `boolean` **`stopped`** verwendet, um die Schleife des Zeitanzeigethreads von außen zu beenden. Dies funktioniert (zumindest bei allen von den Autoren bislang durchgeführten Tests) einwandfrei, da dieses Attribut im Heap verwaltet wird und damit über die Referenz **`t`** auch vom main()-Thread zugänglich ist.

```
public class Zeitanzeige extends Thread {
   public boolean stopped = false;
   public void run() {
      while( !stopped ) {
         ...
      }
   }
}

public class ZeitanzeigeProgramm {
   public static void main( String[] args ) {
      Zeitanzeige t = new Zeitanzeige();
      t.start();
      IO.promptAndReadString( "Beenden durch Eingabetaste\n" );
      t.stopped = true;
   }
}
```

Leider sieht die Realität etwas anders aus: Ein Thread arbeitet in der Regel nicht direkt auf dem Heap sondern auf einem eigenen Speicherbereich, dem Cache, der mit dem Heap ab und an abgeglichen wird. Wann ein Abgleich zwischen Heap und Chache genau stattfindet, lässt sich im Allgemeinen nicht vorhersagen. Daher ist es nicht ausgeschlossen, dass die durch den main()-Thread veranlasste Änderung von `stopped` vom Zeitanzeigethread erst später - oder überhaupt nicht - wahrgenommen wird.

Das Java Memory Model macht diesbezüglich u.a. jedoch die folgenden Zusicherungen:

1. **`volatile`** deklarierte Attribute werden sofort ***im Heap*** abgeglichen
2. wird ein synchronized-Block betreten, kopiert der Thread die Werte aus dem Heap in den Cache
3. wird ein synchronized-Block verlassen, gleicht der Thread seine Änderungen mit dem Heap ab

Ein Lösung des Problems besteht nun einfach darin, das Attribut folgendermaßen zu deklarieren:

```
public volatile boolean stopped = false;
```

Konsultieren Sie vertiefende Literatur zu diesem Thema (etwa [ESS05] S. 193ff. oder [BLO04] S. 189ff.) und geben Sie eine alternative Lösung ohne das Schlüsselwort `volatile` an.