Lösungsansatz Aufgabe 2 FindPrimeNumbers als Runnable

Die Klasse Find `PrimeNumbers` soll `Runnable` implementieren:

```
public class FindPrimeNumbers implements Runnable {
…

@Override
public void run() {
    doCalculations();
}
```

Die Methode `doCalculations()` wird als `private` gekennzeichnet, so dass kein direkter Zugriff mehr möglich ist.

Lösungsansatz Aufgabe 3 Single Thread und Multi Thread Executors

Der Single Thread Executor erhält ein `Runnable` als Aufgabe übergeben und führt sie direkt aus. Es wird kein neuer `Thread` erzeugt. Es handelt sich also um „normale“ Methodenaufrufe wie man es bei der Singlethread-Entwicklung gewohnt ist:

```
public class SingleThreadExecutor implements Executor {

	@Override
	public void execute(Runnable task) {
		task.run();

	}
}
```

Der Multi Thread Executor erzeugt für jede Aufgabe einen neuen Thread und übergibt dem Thread das Runnable. Jede Aufgabe wird nebenläufig bearbeitet.

```
public class MultiThreadExecutor implements Executor {

	@Override
	public void execute(Runnable task) {
		Thread t = new Thread (task);
		t.start();
	}
}
```

Lösungsansatz Aufgabe 4 Thread Pool

Es sollte kein fertiger Executor verwendet, sondern ein eigener Threadpool angelegt werden.

Der Executor benötigt im Wesentlichen zwei Dinge:

- Einen „**Eingangsordner**“ für Aufgaben, die erledigt werden sollen wir verwenden hier eine `BlockingQueue`
- **Threads**, die fortlaufend im Eingangsordner nachschauen, ob neue Aufgaben vorliegen und diese dann sofort abarbeiten. Wenn ein Thread eine Aufgabe abgearbeitet hat, holt er sich die nächste. Er läuft also endlos weiter und schaut immer nach weiteren Aufgaben.

Die `BlockingQueue` legen wir als private Eigenschaft der Klasse an:

```
private final BlockingQueue <Runnable> taskQueue =
                    new ArrayBlockingQueue <Runnable> (NTHREADS * 50);
```

Es ist sinnvoll, die Größe der `BlockingQueue` viel größer als die Anzahl der Threads zu wählen. Wenn Sie vier Büromitarbeiter haben, dann können diese nacheinander Aufgaben abarbeiten. Da jeder Arbeiter nur eine Aufgabe zurzeit bearbeiten kann, sollte der Eingangsordner groß genug sein, damit Aufgaben gepuffert werden können.

Da wir eine fixe Anzahl von Threads verwenden, können diese gleich im Konstruktur des Executors erzeugt werden:

```
public ThreadPoolExecutor () {

      // Erzeuge Threads
      for (int i = 0; i < NTHREADS ; i++) {
            createThread ();
      }
}
```

Die Methode `createThread()` erzeugt nun Threads, die darauf warten, eine neue Aufgabe abzuarbeiten. Dazu wird jeder Thread sofort gestartet und läuft dann in einer Endlosschleife. Der Thread schaut bei jedem Durchgang nach, ob eine neue Aufgabe im „Eingangsordner“ liegt.

```
private void createThread () {
      Thread t = new Thread (){

            public void run () {
                  while (true) {
                        try {

                              Runnable task = taskQueue.take();
                              task.run();
                        } catch (InterruptedException e) {
                              e.printStackTrace();
                        }
                  }

            }     // end method run()
      };          // end new Thread()

      t.start();

}                 // end method createThread()
```

Wichtig sind hier folgende Dinge. Der Thread läuft in einer Endlosschleife. Eine richtiger Executor sollte eine Schnittstelle bereitstellen, um den Service zu stoppen – dies schenken wir uns im Beispiel.

In der Endlosschleife wird fortlaufend eine neue Aufgabe aus dem Eingangsordner genommen:

```
Runnable task = taskQueue.take();
```

Wenn die `taskQueue` gerade leer ist, dann blockiert die `BlockingQueue` den Thread, d.h. er läuft nicht im Leerlauf und verbraucht keine Prozessorleistung. Er wird erst wieder angestoßen wenn tatsächlich ein `Runnable` in der `taskQueue` liegt. Wenn mehrere `Runnables` in der `taskQueue` liegen wird sofort genau eine Aufgabe geliefert. Die Aufgabe wird dann abgearbeitet mit `task.run();`

Da `take()` blockiert wenn gerade keine Aufgaben vorliegen, muss die Anweisungen von einem `try…catch` Block umschlossen werden. Denn es kann sein, dass explizit eine Unterbrechung von außen angefordert wird (z.B. um den Service zu stoppen).

Wir haben jetzt schon fleißige Threads, die auf neue Aufgaben warten. Doch wo kommen die Aufgaben her?

Dazu müssen wir die `taskQueue` füllen. Dies geschieht in der `execute()` Implementierung. Während wir beim Single Thread Executor jede Aufgabe sofort im gleichen Thread ausgeführt haben und beim Multi Thread Executor für jede Aufgabe einen neuen Thread erzeugt haben, müssen wir beim Thread Pool nur die Aufgaben in den Eingangsordner `taskQueue` legen. Die Arbeiter-Threads haben wir ja schon erzeugt. Daher sieht die Implementierung so aus:

```
public void execute(Runnable task) {

	try {
		taskQueue.put(task);
	} catch (InterruptedException e) {
		e.printStackTrace();
	}

}
```

Auch hier müssen wir auf `InterruptedExceptions` reagieren können, da `put()` unter Umständen auch blockiert und zwar immer dann wenn der Eingangsordner voll ist. Sobald dann ein Thread eine Aufgabe herausnimmt und ein Platz frei wird, kommt `put()` zum Zuge. Wenn ohnehin Plätze frei sind, dann legt `put()` die neue Aufgabe einfach direkt in die `taskQueue`. Deshalb war es auch wichtig, dass wir die `taskQueue` etwas großzügiger mit freiem Platz ausgestattet haben.

Lösungsansatz Aufgabe 3 und 4 für die Ausgabe der berechneten Ergebnisse

Beim Multi Thread Executor und Thread Pool Executor werden die Primzahlen nebenläufig berechnet. Das heißt es werden neue Ablaufstränge initiiert, die neben dem main-Thread her laufen. Nun haben wir das Problem, dass der main-Thread parallel weiterläuft und mit `System.out.println(pn)` das Ergebnis ausgeben möchte bevor dieses vorliegt. Somit kommt es zu einer Fehlermeldung.

Es gibt verschiedene Lösungsansätze. Eine Möglichkeit besteht darin, sich eine Liste mit allen Aufgaben zu erstellen, die Berechnung anzustoßen und danach auf die Ergebnisse zu warten:

```
final ArrayList <FindPrimeNumbers> results = new ArrayList<FindPrimeNumbers>();
```

Nach dem die Aufgabe an einen Executor übergeben wird, erfolgt nicht mehr die Ausgabe direkt sondern die Aufgabe wird `results` hinzugefügt:

```
executor.execute(pn);
results.add(pn);
```

Wenn dann alle Berechnungen angestoßen sind, können wir mit einem `Iterator` über alle Resultate gehen und für jedes Resultat warten bis es verfügbar ist:

```
Iterator<FindPrimeNumbers> i = results.iterator();
while (i.hasNext()) {
     String s = i.next().waitForResult().toString();
     System.out.println(s);
}
```

Die Methode `waitForResult()` ist eine neue Methode in `FindPrimeNumbers`. Sie blockiert solange wie das Ergebnis noch nicht berechnet ist. Für die Synchronisierung verwenden wir ein `CountDownLatch`. Da wir nur auf das (eine) Ergebnis pro Aufgabe warten, initialisieren wir den Countdown auf 1:

```
    CountDownLatch latch = new CountDownLatch(1);
```

Am Ende von `doCalculations`[1] zählen wir den Countdown herunter und geben die Barriere somit frei:

```
       latch.countDown();
```

Die Methode waitForResult blockiert nun so lange bis das Latch frei ist (also der CountDown auf 0 gesetzt ist):

```
       public FindPrimeNumbers waitForResult () {
              try {
                     latch.await();
              } catch (InterruptedException e) {
                     e.printStackTrace();
              }
              return this;
       }
```

`latch.await()` blockiert bis der Countdown auf 0 gesetzt ist, daher muss wiederum auf eine `InterruptedException` eingegangen werden – damit das Warten ggs. Unterbrochen werden kann, z.B. beim Abbrechen einer Aktion durch den Benutzer.

[1] Dort wo viele das System.out.println(…) direkt hingeschoben haben. Dies ist auch eine Lösung, allerdings erscheinen die Primzahlen dann nicht mehr geordnet in der Ausgabe.