Lösungen zu den Übungen Concurrency–Execution

1. Concurrency 1 — Java Threads

1.1. Theoretische Fragen [TU]

a. Im Unterricht haben Sie zwei Varianten kennengelernt, um Threads zu erzeugen. Erläutern Sie jeweils, was für die Implementation spezifisch ist und wie die Thread-Instanz erzeugt und gestartet wird.

Variante 1 – von Thread ableiten

Für den parallel laufenden Code wird eine Klasse erstellt, welche von `java.lang.Thread` abgeleitet ist und die run() Methode überschrieben. Zum Starten kann diese direkt erzeugt und gestartet (`start()`) werden.

```
class MyThread extends Thread {
    @Override
    public void run() {
        // do something
    }
}

Thread thread = new MyThread();
thread.start();
```

Variante 2 – Implementieren von Runnable

Für den parallel laufenden Code wird eine Klasse erstellt, welche das Interface `java.lang.Runnable` erfüllt und die `run()` Methode implementiert. Zum Starten wird der Thread mit einer Instanz der Klasse initialisiert, deren `run()` Methode ausgeführt werden soll.

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        // do something
    }
}
Thread thread = new Thread(new MyRunnable());
thread.start();
```

Vorteil: Die Klasse könnte noch von einer anderen (eigenen) Klasse abgeleitet sein.

b. Erläutern Sie im nachfolgenden (vereinfachten) Thread-Zustandsmodell, was die aufgeführten Zustände bedeuten und ergänzen Sie die Mechanismen, welche den Wechsel zwischen den Zuständen auslösen. Wenn vorhanden, geben Sie den entsprechenden Befehl an.

NEW

Thread wurde mit `new` erzeugt, aber noch nicht gestartet.

RUNNABLE

Thread wurde gestartet und wird von der JVM ausgeführt. Die zwei Unterzustände zeigen, ob ein Thread im Moment vom Scheduler aktiviert wurde:

- **Ready**: Thread ist bereit zu arbeiten, wartet aber noch auf die CPU, welche ggf. durch einen anderen Thread belegt ist.
- **Running**: Der Thread wurde vom Scheduler einer CPU zugewiesen und arbeitet

Suspended

Der Thread ist aus diversen Gründen unterbrochen:

- **BLOCKED**: Warten auf monitor bzw. mutex lock
- **WAITING**: Warten auf eine Condition (`wait()` resp. `await()`) oder andere blockierende calls (`join()`)
- **TIMED_WAITING**: Warten auf blockierende Calls mit timeout (`sleep()`, `join(time)`, `wait(time)`)

TERMINATED

Thread wurde beendet (Rückkehr aus `run()`), jedoch ist die Instanz immer noch vorhanden. Sie kann nicht wiederverwendet werden.

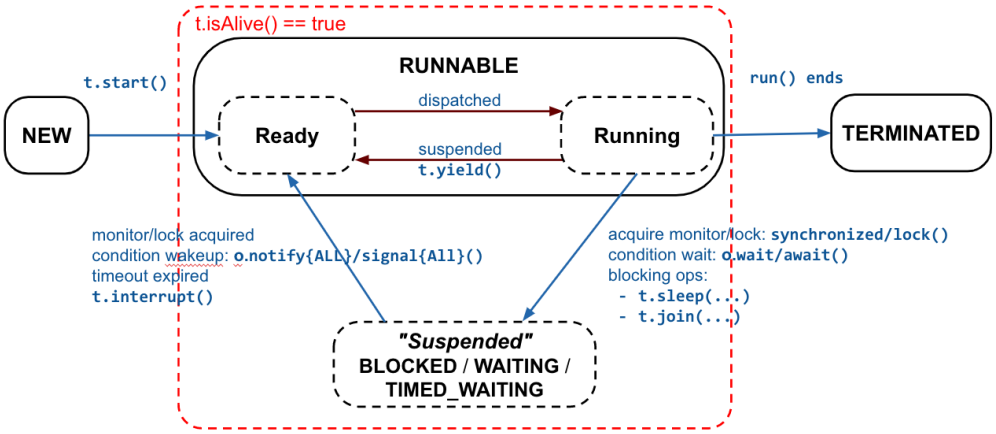

Figure 1. Thread Zustandsmodell Lösung (vereinfacht, mit Übergängen)

1.2. Printer-Threads: Verwendung von Java Threads [PU]

Nachfolgend einige Basisübungen zum Starten und Stoppen von Threads in Java.

```
public class Printer {

    // test program
    public static void main(String[] arg) {
        PrinterThread a = new PrinterThread("PrinterA", '.', 10);
        PrinterThread b = new PrinterThread("PrinterB", '*', 20);
        a.start();
        b.start();
        b.run(); // wie kann das abgefangen werden?
    }


    private static class PrinterThread extends Thread {
        char symbol;
        int sleepTime;

        public PrinterThread(String name, char symbol, int sleepTime) {
```

```
            super(name);
            this.symbol = symbol;
            this.sleepTime = sleepTime;
        }

        public void run() {
            System.out.println(getName() + " run started...");
            for (int i = 1; i < 100; i++) {
                System.out.print(symbol);
                try {
                    Thread.sleep(sleepTime);
                } catch (InterruptedException e) {
                    System.out.println(e.getMessage());
                }
            }
            System.out.println('\n' + getName() + " run ended.");
        }
    }
}
```

a. Studieren Sie das Programm `Printer.java`: Die Methode `Thread.run()` ist public und kann daher direkt aufgerufen werden. Erweitern Sie die Methode `run()` so, dass diese sofort terminiert, wenn sie direkt und nicht vom Thread aufgerufen wird.

   Siehe Code: `PrinterLsgA`

   `Thread.currentThread()` liefert den aktuellen Thread zurück. Wenn die aktuelle Instanz (`this`) nicht mit dem aktuellen Thread übereinstimmt, dann läuft `run()` in einem anderen Thread und wurde direkt aufgerufen. **Achtung**: Vergleichen Sie nicht den Namen, da dieser beim Erzeugen beliebig gesetzt werden kann und auch nicht verhindert wird, dass ein Name doppelt vorkommt.

b. Erstellen sie eine Kopie von `Printer.java` (z.B. `PrinterB.java`) und schreiben Sie das Programm so um, dass die run-Methode über das Interface `Runnable` implementiert wird.

   Führen Sie dazu eine Klasse `PrinterRunnable` ein, die das Interface `Runnable` implementiert. Starten Sie zwei Threads, sodass dieselbe Ausgabe entsteht wie bei (a). Dabei lassen Sie die Erkennung, ob `run()` direkt aufgerufen wurde, weg.

   Siehe Code: `PrinterLsgB`

c. Entfernen Sie die `sleep`-Anweisung. Typischerweise wird jetzt der eine Tasks komplett ausgeführt und dann der andere Task. Wie kann nun erreicht werden, dass die Fairness erhöht wird, d.h. dass der Wechsel zwischen den Threads häufiger erfolgt?
   Hinweis: Je nach Rechner kann der Effekt nicht immer sichtbar sein.

   Siehe Code: `PrinterLsgCD`

   Mit `yield()` kann dem Scheduler signalisiert werden, dass der Thread die CPU abgeben möchte. Der Scheduler kann, muss das Angebot jedoch nicht annehmen. Im Gegensatz zu `pause()` (mit time > 0) welches den Thread immer pausiert, wird ein Wechsel somit nicht erzwungen. Man überlässt dem Scheduler die Entscheidung. Normalerweise finden dann weniger Thread-Wechsel statt. Inwieweit das "fairer" ist, kann diskutiert werden.

d. Wie muss man das Hauptprogramm anpassen, damit der Main-Thread immer als letzter Thread endet?

   Siehe Code: `PrinterLsgCD`

   Mit `join()` kann auf einen bestimmten Thread gewartet werden. Falls der Thread bereits beendet ist, kehrt die Funktion sofort zurück. Wenn man nur darauf wartet, dass alle Threads beendet sind, spielt die Reihenfolge der `join()` Befehle keine Rolle.

2. Concurrency 2 — Executor Framework, Callables and Futures

2.1. Theoretische Fragen [TU]

Im Unterricht haben sie verschieden Arten von Thread-Pools kennengelernt. Welcher davon würde sich für den jeweiligen folgend Anwendungsfall am besten eignen?
Wenn nötig, geben Sie auch die Konfiguration des Thread-Pools an.

a. Sie schreiben einen Server, der via Netzwerk Anfragen erhält. Jede Anfrage soll in einem eigenen Task beantwortet werden. Die Anzahl gleichzeitiger Anfragen schwankt über den Tag verteilt stark.

> **`CachedThreadPool`**
>
> Bei steigender Benutzerzahl wird die Anzahl Threads automatisch erhöht. Sobald diese nicht mehr benötigt werden, werden sie wieder reduziert.

b. Ihr Grafikprogramm verwendet komplexe Mathematik um von hunderten von Objekten die Position, Geschwindigkeit und scheinbare Grösse (aus Sicht des Betrachters) zu berechnen und auf dem Bildschirm darzustellen.

> **`FixedThreadPool` mit Grösse gleich Anzahl CPU-Cores**
>
> CPU intensive Tasks können somit mit optimaler Geschwindigkeit ohne Unterbrechung (Thread-Switching) abgearbeitet werden.

c. Je nach Datenset sind unterschiedliche Algorithmen schneller in der Berechnung des Resultats (z.B. Sortierung). Sie möchten jedoch in jedem Fall immer so schnell wie möglich das Resultat haben und lassen deshalb mehrere Algorithmen parallel arbeiten.

> **`FixedThreadPool` mit Grösse gleich der Anzahl paralleler Algorithmen und Verwendung von `invokeAny`**
>
> Die Algorithmen laufe parallel. Sobald der erste das Resultat liefert, werden die anderen automatisch beendet.

2.2. Prime Checker [PU]

In dieser Aufgabe üben sie die Verwendung des Java Executor Frameworks zum Ausführen von mehreren unabhängigen Aufgaben (Tasks). Mit der Wahl des Typs und der Konfiguration des ExecutorServices, bestimmen Sie auch ob und wie diese Tasks parallel d.h. in Threads ablaufen.

Im Praktikumsverzeichnis finden sie das Modul `PrimeChecker`. Die Anwendung testet für eine Menge an zufälligen grossen Zahlen, ob es sich dabei um eine Primzahl handelt, indem es Brute-Force nach dem kleinstmöglichen Faktor (>1) sucht, durch den die Zahl ganzzahlig geteilt werden kann.

Die Klasse 'PrimeChecker' enthält die Hauptanwendung, welche in einer Schleife zufällige Zahlen erzeugt und testet. Die Verifizierung, ob es sich um eine Primzahl handelt, ist in die Klasse `PrimeTask` ausgelagert, welche bereits `Runnable` implementiert. In der ausgelieferten Form wird jedoch alles im `main`-Thread ausgeführt.

a. Studieren und testen Sie `PrimeChecker`.
Wie lange dauert die Analyse der Zahlen aktuell?

> Die Laufzeit wird am Ende ausgegeben. Je nach Zufallszahl und Rechnerausstattung variiert die Zeit. Sie sollte jedoch im Bereich von rund einer halben Minute liegen.

b. Erweitern Sie `PrimeChecker` damit für jede Analyse (`PrimeTask`-Instanz) mit `new` ein eigener Thread gestartet wird.
   1. Wie lange dauert die Analyse jetzt?

> Siehe Musterlösung Klasse `PrimeChecker`.
> Damit sie die Laufzeit korrekt messen können, müssen Sie die Threads zwischenspeichern und am Schluss warten bis alle beendet sind → `join()`.
> Die Laufzeit sollte sich stark reduzieren, auf zirka ein Drittel der Zeit.

2. Wie viele Threads werden gestartet?

> Für die Berechnung der Primzahlen werden so viele Threads gestartet wie sie Primzahlen berechnen; also 500.

Im nächsten Schritt soll für das Ausführen der `PrimeTask`-Instanzen ein ExecutorService verwendet werden.

c. Ergänzen Sie die Klasse `PrimeCheckerExecutor` so, dass für das Thread-Management jetzt vom ExecutorService erledigt wird. Als Unterstützung sind entsprechende `TODO:` Kommentare enthalten.

1. Welche(r) Thread-Pool-Typ(en) eignet sich für diese Aufgabe?

> Da es sich um CPU-intensive Tasks handelt, eignen sich Pools mit einer fixen Grösse. Am besten ein FixedThreadPool.

2. Wie gross sollte der Thread-Pool sein, um das beste Ergebnis zu erzeugen?
Testen Sie mit unterschiedlichen Pool-Typen und Grössen.

> Die Anzahl Threads sollte die Menge der CPU-Kerne nicht überschreiten, je nachdem, ob noch anderen CPU-intensiven Workloads auf dem Rechner laufen.
> Die Anzahl Kerne können sie von der Laufzeitumgebung abfragen mit:
> `Runtime.getRuntime().availableProcessors();`
> Beim `FixedThreadPool`, wird automatisch die Anzahl CPU-Kerne verwendet, wenn nichts angegeben wird.

d. Stellen Sie sicher, dass der `ExecutorService` am Schluss korrekt heruntergefahren wird.

> Nach dem Auslösen des Herunterfahrens mit `shutdown()`, muss der Main-Thread warten, bis alle Tasks im Pool abgearbeitet sind. Dies kann mit der Methode `executor.awaitTermination(` *`long timeout, TimeUnit unit`* `)` erreicht werden. Wählen Sie den Timeout lang genug (1 Minute sollte reichen);

1. Wie viele Threads werden jetzt gestartet?

> Bei einem `FixedThreadPool` werden für die Berechnung so viele Threads gestartet, wie sie konfiguriert haben. Wird nichts angegeben, wird die Anzahl CPU-Kerne verwendet.
> Bei einem `CachedThreadPool` hängt es stark davon ab, wie viele Threads gleichzeitig laufen. In jedem Fall wird es weniger sein (Faktor ~= Anzahl CPU-Kerne) als wenn jeder Task im eigenen Thread gestartet wird.

2. Was sehen sie bei den Laufzeiten?

> Die Laufzeiten liege alle relativ nahe beieinander. Die Varianz, welche durch die wechselnde Anzahl Primzahlen des zufälligen Samples generiert wird, ist höher als der Unterschied zwischen den verschiedenen Executors. Man müsste ein fixes oder umfangreicheres Sample verwenden, um grössere Unterschiede festzustellen.

Im Moment wird das Resultat nur auf der Konsole ausgegeben, da `Runnable` kein Resultat zurückgeben können. Im nächsten Schritt soll die Anwendung so umgebaut werden, dass die Berechnung in einem Callable passiert und das Resultat im Hauptprogramm verarbeitet (in unserem Fall nur ausgegeben) wird.

e. Ergänzen Sie die Klasse `PrimeTaskCallable` so, dass das Resultat der Berechnung zurückgegeben wird.

Da die Berechnung asynchron erfolgt, können Sie im Hauptprogramm das Resultat nicht mehr so einfach der Zahl zuordnen, für welche die Berechnung gestartet wurde. Deshalb muss im Resultat neben dem Faktor auch die zugehörige Zahl enthalten sein. Dazu können Sie die innere statische Klasse `PrimeTaskCallable.Result` verwenden.

Siehe Musterlösung Klasse `PrimeTaskCallable`.
Stellen Sie sicher, dass sie den generischen Rückgabe-Typ des Callable deklarieren.
`implements Callable<PrimeTaskCallable.Result>`

f. Vervollständigen sie das Hauptprogramm in der Klasse `PrimeCheckerFuture`, welches nun `PrimeTaskCallable` verwenden soll.
Das Resultat soll, wie bei `PrimeChecker`, auf der Konsole ausgegeben werden. Jetzt jedoch im Hauptprogramm.

Beachten Sie, dass das Übermitteln des Tasks an den `ExecutorService` unmittelbar ein Objekt vom Typ `Future` zurückgeliefert, in welchem das Resultat nach Beendigung des Tasks abgelegt wird.
Um auf das Resultat zuzugreifen, ohne die Übermittlung des nächsten Tasks zu blockieren, müssen sie dieses `Future`-Objekt zwischenspeichern (z.B. in einer Liste).
Später können sie die Resultate aus der Liste durchgehen und weiterverarbeiten, was in unserem Fall die Ausgabe auf der Konsole ist.

Siehe Musterlösung Klasse `PrimeTaskCallable`.
Stellen Sie sicher, dass sie den generischen (Rückgabe-)Typ des Callable deklarieren → `implements Callable<PrimeTaskCallable.Result>`

g. Merken Sie einen Unterschied in den Berechnungszeiten oder im Verhalten der Ausgabe? Wenn ja, warum könnte das so sein?

Die Rechenzeiten beim `Callable` sind etwa gleich lang, wie beim `Executor`. Jedoch werden die Resultate jetzt in der Reihenfolge der generierten Primzahlen ausgegeben, da die Liste bei der Ausgabe sequentiell abgearbeitet wird.
Beim `Executor` wurde das Resultat eher nach Rechenzeit (kurze am Anfang) sortiert. Da für Primzahlen praktisch bis zum halben Wert alle Zahlen überprüft werden, dauern die Prüfung länger und die Primzahlen erscheinen eher gegen Schluss.

3. Bewertete Pflichtaufgaben

3.1. Mandelbrot [PA]

Die Lösungen zu den bewerteten Pflichtaufgaben erhalten Sie nach der Abgabe und Bewertung aller Klassen.