Lösungen zu den Pflichtaufgaben Concurrency - Execution

1. Concurrency 1 — Java Threads

Siehe Dokument *Lösungen zu den Übungsaufgaben Concurrency–Execution.*

2. Concurrency 2 — Executor Framework, Callables and Futures

Siehe Dokument *Lösungen zu den Übungsaufgaben Concurrency–Execution.*

3. Bewertete Pflichtaufgaben

3.1. Mandelbrot [PA]

Die JavaFX-Anwendung `Mandelbrot` berechnet die Fraktaldarstellung eines Ausschnitts aus der Mandelbrot-Menge. Dazu wird die zeilenweise Berechnung auf mehrere Threads aufgeteilt.

Sie müssen die Mathematik hinter den Mandelbrotfraktalen nicht verstehen um die Aufgaben zu lösen.

Starten Sie die Anwendung mittels `gradle run` im Verzeichnis `Code/Mandelbrot` bzw. in der IDE mit dem Gradle run task. Es kann sein, dass sie eine Fehlermeldung kriegen, wenn Sie die Mandelbrot-Klasse direkt in der IDE starten (Das ist ein bekanntes JavaFX-Problem).

Im GUI können Sie auswählen, wieviele Threads verwendet werden sollen. Zudem können Sie die Processor-Klasse wählen, die verwendet werden soll. Es gibt 3 Varianten:

- **`MandelbrotTaskProcessor`**: Verwendet ein Array von Worker-Threads die "konventionell" erzeugt und beendet werden. Das Fenster wird in so viel horizontale Bereiche (startRow .. endRow) aufgeteilt, wie Threads zur Verfügung stehen. Jeder Thread berechnet seinen zugewiesenen Zeilenbereich.
  Dieser Processor ist bereits umgesetzt.
- **`MandelbrotExecutorProcessor`**: Hier soll ein `ExecutorService` für das Management der Threads verwendet werden. `MandelbrotTask` soll als `Runnable` implementiert werden, das genau eine Zeile berechnet und diese dem GUI zur Ausgabe übergibt (`processorListener.rowProcessed(row)`). Es müssen also so viele Tasks erzeugt werden, wie das Fenster Zeilen hat (`height`).
  Das Grundgerüst der Klasse ist bereits vorhanden. Der ExecutorService muss ergänzt und `MandelbrotTask`-Klasse angepasst werden.
- **`MandelbrotCallableProcessor`**: Hier soll wiederum ein `ExecutorService` verwendet werden. Diesmal aber soll der `MandelbrotTask` als Callable umgesetzt werden, der jeweils eine Zeile als `Future<ImageRow>` zurückgibt. Diese werden gesammelt und sobald verfügbar

Zeilenweise ans GUI zur Ausgabe übergeben (`processorListener.rowProcessed(row)`).
Das Grundgerüst der Klasse ist bereits vorhanden. Der ExecutorService muss ergänzt und `MandelbrotTask`-Klasse angepasst werden.

Das Thread-Handling ist in die `MandelbrotProcessor`-Klassen im Package `ch.zhaw.prog2.mandelbrot.processors` ausgelagert. Sie müssen nur diese Klassen bearbeiten. Die Benutzeroberfläche und Hilfsklassen sind im übergeordneten Package `ch.zhaw.prog2.mandelbrot` enthalten und müssen nicht angepasst werden.

Analysieren und testen Sie die Anwendung:

a. Mit welcher Anzahl Threads erhalten Sie auf Ihrem Rechner die besten Resultate?

Die Gesamtrechenzeit wird in der Konsole ausgegeben.

In der Regel, erreichen man bei rechenintensiven Tasks das beste Resultat, wenn die Anzahl Threads gleich der Anzahl CPU-Cores ist (`Runtime.getRuntime().availableProcessors()`). Diese wird im GUI automatisch voreingestellt. Das kann jedoch je nach Leistung und Auslastung des Rechners variieren.

b. Wie interpretieren Sie das Resultat im Verhältnis zur Anzahl Cores ihres Rechners?

Die grossen Unterschiede merken sie, wenn sie wenige Threads verwenden, da dann die Parallelität eingeschränkt wird. Ab einer gewissen Anzahl Threads (ca. im Bereich der verfügbaren CPU-Kerne) reduziert sich die benötigte Rechenzeit nicht mehr so stark.
Es kann sogar sein, dass die gesammte Rechenzeit wieder leicht zunimmt, je nach verwendeter MandelbrotProzessor Strategie (siehe nachfolgende Aufgabe)

Aufgabe

Ergänzen Sie die Klassen `MandelbrotExecutorProcessor` und `MandelbrotCallableExecutor`, sowie die jeweiligen inneren Klassen `MandelbrotTask`, so dass diese die obige Beschreibung erfüllen.

Hinweise:

- Die Stellen die angepasst werden müssen sind mit TODO-Kommentaren versehen.
- Überlegen Sie sich, welchen Typ von Thread-Pool hier sinnvollerweise verwendet wird.
- Verwenden Sie den vom Benutzer gesetzten ThreadCount um die Grösse des Thread-Pools zu definieren.
- Neu soll der `MandelbrotTask` nur noch eine Zeile berechnen und ausgeben.
- Überlegen Sie sich welche Optionen Sie haben, um auf die Resultate zu warten und sicherzustellen, dass alle ausgegeben wurden.

Siehe Lösungen in `ch.zhaw.prog2.mandelbrot.processors`:

Die Umsetzung `MandelbrotExecutorProcessor` ist ähnlich wie in der vorherigen Aufgabe `PrimeCheckerExecutor`. Der Executor Service wird in `startProcessing()` erstellt, sowie die Tasks übermittelt und in `stopProcessing()` wird er heruntergefahren. Hier ist es jedoch wichtig, dass sie nicht mit `awaitTermination()` auf die Beendigung warten, da `stopProcessing()` via `taskFinished()` (in der abstrakten Superklasse) vom letzten Task aufgerufen wird und dieser mit `awaitTermination()` selber den shutdown blockieren würde.

Im `MandelbrotCallableProcessor` müssen, wie im PrimeChecker, die vom Executor Service zurückgelieferten Futures in einer Collection-Klasse (z.B. `List`) zwischengespeichert und dann in einem Loop ausgelesen werden, wobei `future.get()` jeweils blockiert bis das Resultat vorhanden ist. Dabei wird immer auf das spezifische Resultat gewartet, obwohl evtl. andere Zeilen schon lange fertig berechnet wären. Es lohnt sich aber in der Regel nicht komplizierte Konstrukte zu bauen, um dies zu optimieren, ausser sie haben wirklich Tasks die sehr lange dauern. Da die Abarbeitung der Futures über die Liste kontrolliert wird, kann am Schluss auch `stopProcessing()` direkt aufgerufen werden und das Konstrukt mit dem Runterzählen der Tasks in `taskFinished()` wird nicht benötigt.

In der Musterlösung haben wir noch zwei weitere Varianten hinzugefügt:

`MandelbrotCompletionService` löst das Problem mit der linearen Abarbeitung der Futures. Der `CompletionService` verfolgt die Zustände der zurückgelieferten Futures selbständig und liefert mit `take()` das das nächste Future, dessen Resultat verfügbar ist. Falls kein Future completed ist, blockiert der Aufruf, bis ein Resultat vorhanden ist.
Bei so kurzen Tasks ist der Gewinn gegenüber dem sequenziellen Abarbeiten der Liste klein.

`MandelbrotCompletableFutureProcessor` verwendet zur Abarbeitung sogenannte `CompletableFutures`. Dabei handelt es sich um eine Erweiterung von `Future`, welche es erlaubt ganze Abläufe von Schritten (Pipelines) zu definieren, die dann jeder für sich asynchron ausgeführt werden können. In unserem Fall wird, sobald das Resultat der Berechnung einer Zeile vorhanden ist, die Methode `processorListener.rowProcessed(imageRow) aufgerufen und, sobald diese beendet ist, `taskFinished()`. Da das Resultat jeweils direkt von Schritt zu Schritt weitergereicht wird, muss es nicht in einer Liste zwischengespeichert werden.
In diesem Beispiel verwenden wir, wegen der besseren Lesbarkeit, Lambda-Ausdrücke, welche im Themenbereich Functional Programming behandeln werden. Diese könnten auch durch anonyme innere Klassen vom Typ `java.util.function.Consumer` ersetzt werden.