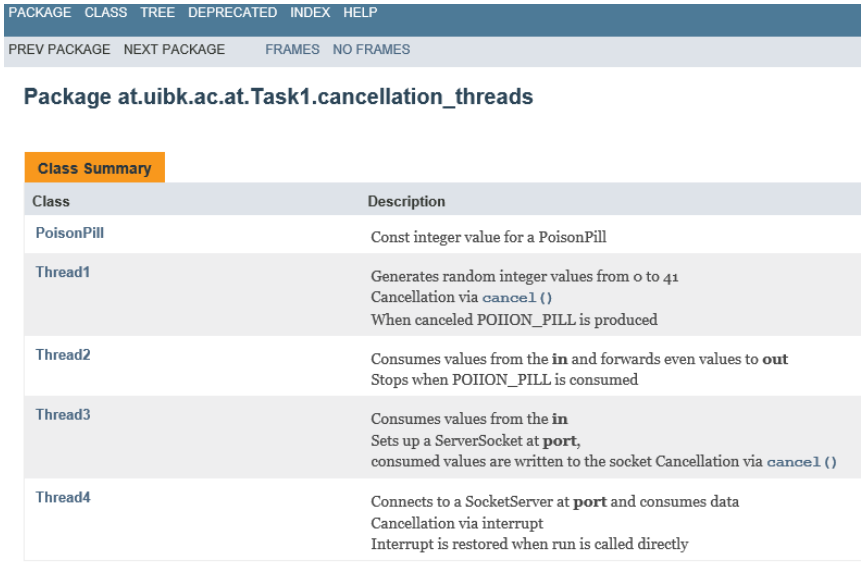
Florian Tischler
Mat. Nr.: 1315066
Gruppe: 6
Gruppenmitglieder: Tischler Florian, Mathias Hölzl
gelöste Aufgaben: 3/3
resultierende Punkte: 100%

Aufgabe 1

Für die Kommunikation zwischen Thread 1,2,3 wurde eine Blockingqueue verwendet. Die Beschreibung für C.) war jedoch etwas unklar. Bei C.c „Vorgabe: die Abarbeitung des Sockets dauert zu lange und muss händisch abgebrochen warden" wussten wir leider nicht was damit gedacht wäre. Unsere Lösung ist, dass mittels Console (ENTER) zufällig lange Daten von Thread 3 and Thread 4 gesendet werden. Gibt man in die Console exit ein so wird Thread 1 gestoppt. Thread 1 sendet daraufhin die PoisionPill an Thread2. Thread 3 wird erst beendet wenn Thread 1 und 2 beendet sind. Thread 4 wird als letzter mit einen interrupt gestoppt.

Dokumentation:
Die generierten html datein befinden sich unter ./doc/. (Einfach ./doc/index.html im Browser öffnen)
Ausschnitt index.html:



PACKAGE CLASS TREE DEPRECATED INDEX HELP

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Package at.uibk.ac.at.Task1.cancellation_threads

Class Summary

Class	Description
PoisonPill	Const integer value for a PoisonPill
Thread1	Generates random integer values from 0 to 41 Cancellation via `cancel()` When canceled POIION_PILL is produced
Thread2	Consumes values from the **in** and forwards even values to **out** Stops when POIION_PILL is consumed
Thread3	Consumes values from the **in** Sets up a ServerSocket at **port**, consumed values are written to the socket Cancellation via `cancel()`
Thread4	Connects to a SocketServer at **port** and consumes data Cancellation via interrupt Interrupt is restored when run is called directly

Aufgabe 2

Anmerkungen zur Implementierung:

Um immer den besten Job zu stehlen wird aus der Deque jedes anderen Rechenzentrums der letzte Job gestohlen. Dann wird ermittelt welcher Job der Beste ist und die anderen werden zurückgegeben. Die Implementierung kommt ohne die besonderen Methoden der BlockingDeque aus – mir war nicht klar wofür diese nötig gewesen wären.

Aufgabe 3

e)

FutureTask könne überall dort verwendet werden wo das Ergebnis einer asynchronen Operation benötigt wird (z.B.: Divide & Conquer Algorithmen). Hier liegt der Vorteil darin, dass der Zugriff auf das Result des Subproblems einfach über das Future-Objekt zugegriffen werden kann und keine manuelle Synchronisierung nötig ist.

Mithilfe von FutureTask können auch leicht Asynchrone Funktionsketten erstellt werden. Hat man zum Beispiel drei Funktionen die alle asynchron ausgeführt werden sollen. Die ersten zwei werden sofort gestartet und die dritte soll erst beginnen sobald sie die Ergebnisse der anderen beiden erhalten hat. Hier übergibt man einfach der dritten Funktion die Future-Objekte der ersten zwei Tasks und diese holt wartet dann bis die ersten beiden fertig sind.

Ein weiterer Anwendungsfall sind die JavaFX Tasks welche vom FutureTask<T> abgeleitet.

Anmerkungen zur Implementierung:

Durch ungenaues Lesen wurde die CalculatePartOfPi Klasse nicht mittels FutureTask<T> erstellt sondern durch implementieren der RunnableFuture<T> Schnittstelle. Da FutureTask<T> jedoch ebenfalls RunnableFuture<T> implementiert denke ich, dass dies kein Problem darstellen sollte.

Bei der Messung der Ausführungszeiten ist es in manchen Fällen so, dass die Zeit ohne Instanziierung der Objekte größer als die Zeit mit. Ich führe dieses Verhalten auf den Umstand zurück, dass ich die Funktion die Pi berechnet etwas umschreiben musste da die Zeit für das Erstellen der CalculatePartOfPi Objekte sonst nicht möglich gewesen wäre (siehe Klasse Pi und deren Funktionen calcAndProfile() und calcAndProfileWithoutInstantiation())

Anmerkungen zu den Vorlesungsfolien:

Auf der Folie 20 befindet sich ein Fehler bezüglich der Future Klasse: diese implementiert nicht Runnable. Erstens weil Future selbst eine Schnittstelle ist und zweitens weil es hierfür die RunnableFuture Schnittstelle gibt. Wurde hier vielleicht FutureTask anstatt Future gemeint?