
Artemis 6.4.0

Course Overview

ge64baw

Courses > Praktikum: Grundlagen der Programmierung WS22/23 > Exercises > W08H02 - Effiziente Aufgabenverwaltung

W08H02 - Effiziente Aufgabenverwaltung

Hausaufgabe



Medium

Submission due: 8 months ago

Complaint due: 7 months ago


Points: 6 of 6


Assessment: manual ?

100%  (8 months ago)

GRADED


Recent results:




0% 

(8 months ago)


GRADED




0% 

(8 months ago)


GRADED




100% 

(8 months ago)


GRADED



100% 

(8 months ago)

GRADED

Show all results 

Tasks:

Effiziente Aufgabenverwaltung


Die PUM ist eine weltweit agierende Institution. Daher lässt es sich auch nicht vermeiden, dass einige Aufgaben mehrfach ausgeführt werden müssen. Aber auch Pinguine müssen Strom sparen. Deshalb haben sich die Forschuine dazu entschieden, ein System aufzubauen, mit dem sich mehrfache Berechnungen (und das mehrfache Abspeichern dieser) vermeiden lassen. Das System haben sie schon durchgeplant, jedoch haben sie wichtige Deadlines für eine anstehende Konferenz in Südgeorgien. Da Pinguine nicht fliegen können, müssen sie den ganzen Weg schwimmen, müssen also rechtzeitig aufbrechen und alles vorher fertig haben. Deswegen liegt es wieder an dir, die Implementierung zu übernehmen.

Hinweise

- Bei dieser Aufgabe gibt es keine Public Behavior Tests. Lediglich die Anwesenheit von geforderten Attributen und Methoden, deren Rümpfe nicht vorgegeben sind, werden öffentlich getestet.
- Es gibt keine Einschränkungen bzgl. Bibliotheken, die verwendet werden dürfen. Ebenso dürfen private Hilfsmethoden, Attribute und Klassen verwendet werden.
- Beispiele zu erwarteten Verhaltensweisen findest du in den `main`-Methoden des Templates oder am Ende der Aufgabenstellung.


Aufgabe

In dieser Aufgabe geht es darum, ein System für `Tasks` aufzubauen, dass ähnlich wie Javas `String`-Pool funktioniert (siehe z.B. [hier](#)). Dabei soll jeweils nur eine äquivalente `Task` gleichzeitig gespeichert werden. Um dies zu erreichen, werden `Tasks` in einem `TaskPool` gespeichert und verwaltet und über eine `TaskFactory` initialisiert.

 **TaskFunction** [1 of 1 tests passing](#)

Die Klasse `TaskFunction` dient als Wrapper für eine Java `Function`, die mit einer eindeutigen `ID` identifiziert werden kann. Da du `FunctionalInterfaces` vielleicht noch nicht kennst, haben dir die Forschuine eine Auswahl an `Functions` in der utility Klasse `FunctionLib` mitgegeben, damit du mit diesen deine Implementierung testen kannst, ohne dir Gedanken über die Deklaration machen zu müssen. Evaluieren kannst du eine `Function` mit `function.apply(input)`.
`TaskFunction` hat folgende Anforderungen:

- Attribute `int ID` und `Function<T,R> function`, die jeweils `private` und `final` sein sollen.
- Einen Konstruktor, der als Parameter eine `Function<T,R>` erwartet und das Attribut mit dem entsprechenden Wert initialisiert (du darfst davon ausgehen, dass nie `null` übergeben wird). Außerdem soll `ID` mit einer fortlaufenden ID initialisiert werden, beginnend bei `0`. Das heißt, dass bei jedem Programmstart die erste Instanz die ID `0` erhalten soll, die zweite ID `1`, die dritte ID `2` und so weiter. (Dem Konstruktor kannst du zum Testen einfach einen konstanten Member aus `FunctionLib` mitgeben)
- Eine Methode `public R apply(T)`, die `function` auf den übergebenen Parameter anwendet und das Ergebnis zurück gibt.
- Überschriebene Methoden `hashCode` und `equals`, die jeweils nur von `ID` abhängig sein sollen. Die Anforderungen dieser Methoden kannst du in der `JavaDoc` nachlesen. Zwei `TaskFunctions` sind genau dann gleich, wenn sie dieselbe `ID` haben. **Tipp:** du darfst `Objects.hash()` verwenden.

 **Task** [1 of 1 tests passing](#)

Die Klasse `Task` dient zur Speicherung einer einzelnen Aufgabe bestehend aus Eingabe, Berechnungsfunktion und Ergebnis. Dabei ist wichtig, dass das Ergebnis nur bei Bedarf ("lazy") berechnet wird und auch nur einmal (also genau dann, wenn zum ersten Mal auf den Wert zugegriffen wird, wir wollen ja Rechenpower sparen). Du darfst davon ausgehen, dass das Ergebnis nie `null` sein wird. Die Klasse hat folgende Anforderungen:

- Attribute `T input`, `R result` und `TaskFunction<T,R> taskFunction`, alle sollen `private` sein und Getter haben. Zusätzlich sollen `input` und `taskFunction` `final` sein.
- Einen `protected` Konstruktor, der initiale Werte für `input` und `taskFunction` (in dieser Reihenfolge) erwartet und die Attribute entsprechend initialisiert. Das `result` soll noch **nicht** berechnet werden.

https://artemis.ase.in.tum.de/courses/201/exercises/8699

1/3

- Methoden `hashCode` und `equals`, die jeweils genau von beiden finalen Attributen abhängig sind. Zwei `Tasks` sind also genau dann gleich, wenn sowohl `input`, als auch die `taskFunction` gleich sind.

? TaskPool No tests

Die Klasse `TaskPool` dient als Speicherort der `Tasks` und sorgt dafür, dass jede Aufgabe nur einmal gespeichert werden kann. Es gibt **keinen** Public Test für diese Teilaufgabe!

Die Klasse hat folgende Anforderungen:

- Einen parameterlosen `protected` Konstruktor.
- Eine Methode `Task<T,R> insert(Task<T,R>)`, die die übergebene Aufgabe in den Pool aufnimmt, falls sie noch nicht vorhanden ist, und wieder zurückgibt. Falls schon eine äquivalente Aufgabe enthalten ist, so wird das äquivalente Objekt zurückgegeben. In diesem Fall passiert mit dem Parameter nichts weiter.
- Eine Methode `Task<T,R> getByValue(T, TaskFunction<T,R>)`, die den Pool nach einer `Task` mit zu den Parametern äquivalenten `input` und `taskFunction` Attributen durchsucht und dieses zurückgibt. Sollte keine solche `Task` gefunden werden, so soll `null` zurückgegeben werden.
- Hinweis:** In was für einer Datenstruktur du die `Tasks` verwaltest, bleibt dir überlassen. (Tipp: eine `HashMap` könnte hilfreich sein).

✓ TaskFactory 1 of 1 tests passing

Die Klasse `TaskFactory` dient dazu, neue `Tasks` zu erstellen oder auf diese zuzugreifen, ohne dass dabei Duplikate entstehen. Dafür hat die Klasse folgende Anforderungen:

- Ein `private final` Attribut `TaskPool<T, R> pool`, in dem die `Tasks` gespeichert werden können.
- Einen parameterlosen Konstruktor.
- Eine Methode `Task<T,R> create(T, TaskFunction<T,R>)`, die eine `Task` mit den entsprechenden Attributwerten zurückgibt. Falls eine äquivalente `Task` bereits im Pool enthalten ist, so soll diese zurückgegeben werden. Andernfalls (und nur dann) wird erst eine neue erstellt, in den Pool aufgenommen und dann zurückgegeben.
- Eine Methode `Task<T,R> intern(Task<T,R>)`, die die Pool-Repräsentation der übergebenen `Task` zurückgibt. Falls eine äquivalente `Task` bereits vorhanden ist, so wird diese zurückgegeben, andernfalls wird die übergebene `Task` in den Pool aufgenommen.

Beispiele

In diesem Abschnitt findest du einige Beispiele zur Verwendung der einzelnen Klassen. Diese findest du auch in den main-Methoden des Templates.

▼ 1. TaskFunction

```
public static void main(String[] args) {
    TaskFunction<Integer, Integer> f1 = new TaskFunction<>(FunctionLib.SQUARE);
    TaskFunction<Integer, Integer> f2 = new TaskFunction<>(FunctionLib.SUM_OF_HALFS);
    TaskFunction<Integer, Integer> f3 = new TaskFunction<>(FunctionLib.SQUARE);
    System.out.println(f1.equals(f2)); // false
    System.out.println(f1.equals(f3)); // false
    System.out.println(f1.equals(f1)); // true
    System.out.println(f1.apply(2)); // 4
}
```

`f1` und `f3` berechnen zwar dasselbe, jedoch haben sie verschiedene IDs und werden nicht als gleich angesehen.

▼ 2. Task

```
public static void main(String[] args) {
    TaskFunction<Integer, Integer> f1 = new TaskFunction<>(FunctionLib.INC);
    TaskFunction<Integer, Integer> f2 = new TaskFunction<>(FunctionLib.INC);
    Task<Integer, Integer> t1 = new Task<>(1, f1);
    Task<Integer, Integer> t2 = new Task<>(1, f1);
    Task<Integer, Integer> t3 = new Task<>(1, f2);

    System.out.println(t1.equals(t2)); // true
    System.out.println(t1.equals(t3)); // false

    System.out.println(t1.getResult()); // 2
}
```

Auch hier gilt: `t1` und `t3` berechnen dasselbe, aber mit unterschiedlichen `TaskFunctions`. Daher sind sie nicht gleich. `t1` und `t2` hingegen sind gleich, da sie sowohl `input`, als auch die Funktion teilen. Hätte `t2` eine andere Zahl als `input`, wäre das anders.

▼ 3. TaskPool

```
public static void main(String[] args) {
    TaskFunction<Integer, Integer> f = new TaskFunction<>(FunctionLib.SQUARE);
    TaskPool<Integer, Integer> tp = new TaskPool<>();

    System.out.println(tp.getByValue(1, f)); // null

    Task<Integer, Integer> t1 = new Task<>(1, f);
    Task<Integer, Integer> t2 = new Task<>(1, f);
    System.out.println(t1 == tp.insert(t1)); // true
    System.out.println(t1 == tp.insert(t2)); // true
    System.out.println(t1 == tp.getByValue(1, f)); // true
}
```

t1 und t2 sind equivalen/gleich, daher wird beim 2. insert auch t2 nicht in den Pool aufgenommen, sondern t1 zurückgegeben. Die Kombination aus 1 und f würde eine Task äquivalent zu t1 ergeben, daher gibt getByValue auch t1 zurück.

▼ 4. TaskFactory

```
public static void main(String[] args) {
    TaskFactory<Integer, Integer> tf = new TaskFactory<>();
    TaskFunction<Integer, Integer> f = new TaskFunction<>(FunctionLib.SQUARE);
    Task<Integer, Integer> t1 = tf.create(5, f);
    Task<Integer, Integer> t2 = new Task<>(5, f);
    System.out.println(t1 == tf.create(5, f)); // true
    System.out.println(t1 == tf.intern(t2)); //true
}
```

Das 2. create mit 5 und f würde eine zu t1 äquivalente Task erzeugen und gibt daher direkt t1 zurück. Auch t2 ist äquivalent zu t2, weshalb intern t1 zurück gibt.
Anmerkung: normalerweise würde man t2 natürlich nicht so verwenden (sonst hätten wir uns die Implementierung dieser Aufgabe auch sparen können), Stattdessen würde man eher `Task<Integer, Integer> t2 = tf.intern(new Task<>(5,f));` verwenden oder direkt auf `create` zurückgreifen.

Tests nach der Deadline

- ✔ Punkte Tests 6 of 6 tests passing
Hier werden nach der Deadline die Tests angezeigt, die die Punkte verteilen.

Exercise details	
Release date:	Dec 8, 2022 18:30
Submission due:	Dec 18, 2022 18:00
Assessment due:	Jan 8, 2023 18:00
Complaint due:	Jan 15, 2023 18:00

Every student is allowed to complain once per exercise. In total 1000 complaints are possible in this course. You still have 998 complaints left. ⓘ

How useful is this feedback to you?

★ ★ ★ ★ ★