Funktionale und objektorientierte Programmierkonzepte
Übungsblatt 07

TECHNISCHE UNIVERSITÄT DARMSTADT

Entwurf

Achtung: Dieses Dokument ist ein Entwurf und ist noch nicht zur Bearbeitung/Abgabe freigegeben. Es kann zu Änderungen kommen, die für die Abgabe relevant sind. Es ist möglich, dass sich **alle** Aufgaben noch grundlegend ändern. Es gibt keine Garantie, dass die Aufgaben auch in der endgültigen Version überhaupt noch vorkommen und es wird keine Rücksicht auf bereits abgegebene Lösungen genommen, die nicht die Vorgaben der endgültigen Version erfüllen.

Hausübung 07

Gesamt: -4 Punkte

Lambda-Ausdrücke

-NoValue-

Verbindliche Anforderungen für die gesamte Hausübung:

- Auch in dieser Hausübung fordern wir wieder Dokumentation mittels `JavaDoc`. Informationen dazu finden Sie unter anderem auf Übungsblatt 03.
- Schreiben Sie für die einzelnen Komponenten Ihrer Lösung immer sofort Tests und wenden Sie sie auch sofort zur eigenen Kontrolle an, wie Sie es von den früheren Übungsblättern her gewohnt sein sollten!

Einleitung

In der letzten Hausübung ging es primär um Methodenimplementationen, also Anweisungen und Ausdrücke, davor ging es um Klassen und Interfaces. Jetzt kommen wir zu einem „Schmankerl“, das beides miteinander verbindet: Lambda-Ausdrücke. Das ist eine der vielen schönen und nützlichen programmiersprachlichen Konstrukte, die in funktionalen Sprachen (wie Racket) entwickelt und in andere Programmiersprachen (wie Java) übernommen worden sind. Leider leidet die Schönheit bei einer solchen Übertragung zwangsläufig, ist aber immer noch vielleicht sichtbar. Die Nützlichkeit ist jedenfalls auch in Sprachen wie Java hoch.

Wie in Java üblich, ist jede `public`-Klasse C in einer Quelltextdatei C`.java` und jedes `public`-Interface I analog in einer Quelltextdatei I`.java` zu definieren. Das werden wir ab diesem Übungsblatt nicht mehr explizit dazuschreiben.

Alle Klassen und Interfaces aus H1 sollen im package `h07.predicate`, alle Klassen und Interface aus H2 sollen im package `h07.person` landen. Der letzte Satz ist 1:1 aus 21/22 übernommen → anpassen an das Folgende.

H1: Binäre Operatoren auf `double` als Functional Interfaces -1 Punkte

Mit `DoubleBinaryOperator` ist im Folgenden `java.util.function.DoubleBinaryOperator` gemeint. Für eine Variable oder Konstante `op` vom statischen Typ `DoubleBinaryOperator` mit `op != null` sagen wir bei Aufruf `op.applyAsDouble(`x, y`)` im Folgenden, dass `op` auf x und y ***angewendet*** wird.

H1.1: Erste binäre Operatorklasse auf `double` -1 Punkte

Schreiben Sie eine `public`-Klasse `DoubleSumWithCoefficientsOp`,[1] die das `public`-Interface `DoubleBinaryOperator` implementiert. Der `public`-Konstruktor von `DoubleSumWithCoefficientsOp` hat zwei Parameter vom Typ `double`: `coeff1` und `coeff2` (in dieser Reihenfolge). Passend zu diesen beiden Parametern hat ein Objekt der Klasse `DoubleSumWithCoef- ficientsOp` zwei `private`-Objektkonstanten `coeff1` und `coeff2` vom Typ `double`, die durch die beiden Parameterwerte so initialisiert werden, wie Sie das von früheren Hausübungen kennen. Die Methode `applyAsDouble` von Klasse `DoubleSumWithCoefficientsOp` soll ihren ersten Parameter mit `coeff1` und ihren zweiten Parameter mit `coeff2` multiplizieren und die Summe dieser beiden Produkte zurückliefern.

Beachten Sie, dass dieser binäre Operator nicht kommutativ ist (außer im Fall `coeff1==coeff2`), das heißt, die Reihenfolge der aktualen Parameterwerte von `applyAsDouble` ist signifikant.

H1.2: Zweite binäre Operatorklasse auf `double` -1 Punkte

Schreiben Sie eine `public`-Klasse `EuclideanNorm`, die das `public`-Interface `DoubleBinaryOperator` implementiert. Für ihre aktualen Parameterwerte x und y liefert die Methode `applyAsDouble` den Wert $\sqrt{x^2 + y^2}$ mit Hilfe der Klassenmethode `sqrt` von Klasse `Math` zurück (gemeint ist natürlich `java.lang.Math`).

H1.3: Dritte binäre Operatorklasse auf `double` -1 Punkte

Schreiben Sie eine `public`-Klasse `DoubleMaxOfTwo`, die das `public`-Interface `DoubleBinaryOperator` implementiert. Die Methode `applyAsDouble` dieser Klasse liefert das Maximum ihrer beiden aktualen Parameterwerte zurück.

H1.4: Vierte binäre Operatorklasse auf `double` -1 Punkte

Schreiben Sie eine `public`-Klasse `ComposedDoubleBinaryOperator`, die das `public`-Interface `DoubleBinaryOperator` implementiert. Ein Objekt dieser Klasse hat drei `private`-Objektattribute vom Typ `DoubleBinaryOperator`. Der `public`-Konstruktor hat drei Parameter – `op1`, `op2` und `op3` – vom formalen Typ `DoubleBinaryOperator` und initialisiert damit die drei Attribute auf die übliche Weise.

Eine Anwendung einer Variablen oder Konstanten von `ComposedDoubleBinaryOperator` auf x und y beginnt damit, dass einmal `op1` und separat davon einmal `op2` auf x und y angewendet wird, wobei x jeweils der erste aktuale Parameterwert ist. Auf die Ergebnisse dieser beiden Anwendungen wird dann `op3` angewendet, wobei das Ergebnis von `op1` der erste und das Ergebnis von `op2` der zweite aktuale Parameter für `op3` ist.

Auch dieser binäre Operator ist in der Regel nicht kommutativ.

[1] Siehe z.B. `https://de.wikipedia.org/wiki/Koeffizient` für die Namensgebung.

***Hinweis* (0 Punkte):** Bevor Sie weitergehen, sollten Sie die vier bis jetzt erstellten binären Operatorklassen testen, um sicherzugehen, dass sie korrekt sind und dass Laufzeitfehler bzw. falsche Ergebnisse, die Sie in der Bearbeitung der folgenden Aufgaben bekommen, höchstwahrscheinlich nicht durch diese vier Klassen verursacht sind – dass also der Fehler anderswo zu suchen ist. Unseres Erachtens sollte es dafür ausreichen, dass Sie ein Objekt der vierten Klasse mit je einem Objekt der ersten drei Klassen einrichten und die Korrektheit des Ergebnisses für ein paar gut ausgewählte Zahlenpaare prüfen.

H2: Lambda-Ausdrücke in Kurzform und Standardform in „Operatorenfabrik“ -1 Punkte

Schreiben Sie eine `public`-Klasse `PairOfDoubleCoefficients` mit zwei `public`-Objektattributen `coeff{1,2}` vom Typ `double`.

Schreiben Sie eine `public`-Klasse `TripleOfDoubleBinaryOperators` mit drei `public`-Objektattributen `operator{1,2,3}` vom Typ `DoubleBinaryOperator`.

Schreiben Sie eine `public`-Klasse `DoubleBinaryOperatorFactory`. Die `public`-Klassenmethode `buildOperator` von `Double- BinaryOperatorFactory` hat einen ersten Parameter vom formalen Typ `String`, einen zweiten Parameter vom formalen Typ `Object` (beide aus `java.lang`) und einen dritten Parameter vom formalen Typ `boolean`. Die Methode `buildOperator` gibt nur dann einen Wert ungleich `null` zurück, wenn der erste aktuale Parameter einer der vier Strings `"Coeffs"`, `"Euclidean"`, `"Max"` oder `"Composed"` ist.

Falls der dritte Parameter aktualen Wert `false` hat, wird die Rückgabe von `buildOperator` mit Operator `new` erzeugt, wie Sie es kennen. Im Einzelnen:

- `"Coeffs"`: Falls der dynamische Typ des zweiten aktualen Parameters gleich `PairOfDoubleCoefficients` oder ein Subtyp davon ist, ist der dynamische Typ der Rückgabe `DoubleSumWithCoeffsOp`, und die beiden Koeffizienten werden aus dem zweiten aktualen Parameter geholt. Andernfalls wird `null` von `buildOperator` zurückgeliefert.
- `"Euclidean"`: Der dynamische Typ der Rückgabe ist `EuclideanNorm`.
- `"Max"`: Der dynamische Typ der Rückgabe ist `DoubleMaxOfTwo`.
- `"Composed"`: Falls der dynamische Typ des zweiten aktualen Parameters gleich `TripleOfDoubleBinaryOperators` oder ein Subtyp davon ist, ist der dynamische Typ der Rückgabe `ComposedDoubleBinaryOperator`, und die drei Operatoren werden aus dem zweiten aktualen Parameter geholt. Andernfalls wird `null` von `buildOperator` zurückgeliefert.

Falls hingegen der dritte Parameter aktualen Wert `true` hat, wird die Rückgabe in jedem der vier Fälle durch einen ***Lambda-Ausdruck*** gebildet, Ausdrücke mit Operator `new` sind in diesem Fall nicht erlaubt. Dieser Lambda-Ausdruck ist in jedem der vier Fälle logisch äquivalent zur jeweils oben spezifizierten Rückgabe. Die beiden Lambda-Ausdrücke für `"Coeffs"` und `"Euclidean"` sollen in ***Standardform*** gemäß Folie 89-97 von Kapitel 04c sein. Der Lambda-Ausdruck für `"Composed"` soll in ***Kurzform*** gemäß Folie 89-97 von Kapitel 04c sein.

Speziell bei `"Max"` gibt es in dem Fall, dass der dritte aktuale Parameter Wert `true` hat, noch eine Besonderheit: Falls der dynamische Typ des zweiten aktualen Parameters nicht `Boolean` ist, soll wieder `null` zurückgeliefert werden. Andernfalls soll sein: Falls `true` in diesem `Boolean`-Objekt eingekapselt ist, soll der Lambda-Ausdruck auch hier in ***Kurzform*** gemäß Folie 89-97 von Kapitel 04c sein, wobei hier für die Maximumsberechnung keine Methode aufgerufen wird, sondern das Maximum wird mit Hilfe des Bedingungsoperators berechnet. Falls hingegen `false` in diesem `Boolean`-Objekt eingekapselt ist, soll im Lambda-Ausdruck eine Methodenreferenz mit der Methode `max` von Klasse `Math` verwendet werden (siehe Folien 186 ff. in Kapitel 04c).

***Verbindliche Anforderung*:** Im Falle, dass der dritte Parameter aktualen Wert `true` hat, sollen die fünf(!) verschiedenen Fälle ausschließlich mit `if`-Verzweigungen unterschieden werden, im Falle `false` in einer einzigen `switch`-Anweisung (siehe Kapitel 03c, Folien 214 ff.).

***Verständnisfragen am Rande* (0 Punkte):**

- Sehen Sie hier irgendwo Closure gemäß Folien 68-70 in Kapitel 04c?
- Dieses Design mit einem zusätzlichen Parameter vom Typ `Object` für fallspezifische, optionale Zusatzinformationen ist offensichtlich sehr flexibel. Es wird auch an verschiedenen Stellen in der Java-Standardbibliothek angewandt. Welches Risiko steckt darin?
- Sie lesen oben mehrfach „oder ein Subtyp davon". Warum? Bei einem Parameter vom formalen Typ `String` oder `Boolean` können wir uns gewiss sein, dass `String` bzw. `Boolean` auch der aktuale Typ ist. Warum? Was ist die hilfreiche Konsequenz aus dieser Gewissheit, wenn Sie mit `instanceof` testen wollen, ob der dynamische Typ gleich `String` bzw. `Boolean` ist?

H3: Unäre und binäre Operatoren auf „Array von `double`" als Functional Interfaces -1 Punkte

H3.1: Interfaces für binäre Operatoren auf „Array von `double`" -1 Punkte

Schreiben Sie ein `public`-Interface `DoubleArrayUnaryOperatorGivingArray` mit einer funktionalen Methode `applyAsDou- bleArray`, die einen Parameter vom formalen Typ „Array von `double`" und Rückgabetyp „Array von `double`" hat.

Schreiben Sie ein `public`-Interface `DoubleArrayBinaryOperatorGivingArray` mit einer funktionalen Methode `applyAsDou- bleArray`, die zwei Parameter vom formalen Typ „Array von `double`" und Rückgabetyp „Array von `double`" hat.

Schreiben Sie ein `public`-Interface `DoubleArrayBinaryOperatorGivingScalar` mit einer funktionalen Methode `applyAs- DoubleArray`, die zwei Parameter vom formalen Typ „Array von `double`" und Rückgabetyp `double` hat.

***Verständnisfrage am Rande* (0 Punkte):** In den folgenden drei Teilaufgaben ist von „Filter", „Map" und „Fold" die Rede. Können Sie sich vorstellen, was diese drei Teilaufgaben mit dem zu tun haben, was in Kapitel 04c als „Filter", „Map" und „Fold" bezeichnet wird?

H3.2: Unäre Filter-Klasse auf „Array von `double`" -1 Punkte

Schreiben Sie eine `public`-Klasse `ReduceDoubleArray`, die das `public`-Interface `DoubleArrayUnaryOperatorGivingArray` implementiert. Ein Objekt dieser Klasse hat ein `private`-Objektattribut vom Typ `DoublePredicate` (gemeint ist natürlich `java.util.function.DoublePredicate`). Der `public`-Konstruktor hat einen Parameter vom selben formalen Typ und initialisiert das Attribut damit wie üblich.

Falls einer der beiden aktualen Parameterwerte der Methode `applyAsDoubleArray` gleich `null` ist, liefert diese Methode `null` zurück. Andernfalls liefert sie ein Array zurück, das höchstens so lang ist wie der aktuale Parameter

(die Länge der Rückgabe kann auch 0 sein). Konkret liefert sie ein Array zurück, das bestimmte Komponenten aus dem aktualen Parameter enthält, nämlich alle Komponenten, für die Methode `test` des Prädikats `true` liefert. Diese Komponenten sind in der Rückgabe in derselben Reihenfolge wie im aktualen Parameter. Darüber hinaus hat die Rückgabe keine weiteren Parameter.

Ihre Methode `applyAsDoubleArray` geht zweimal durch den aktualen Parameter: einmal, um die Länge der Rückgabe zu bestimmen, und ein zweites Mal, um die Komponenten des zurückzuliefernden Arrays zu setzen.

H3.3: Binäre Map-Klasse auf „Array von `double`“ -1 Punkte

Schreiben Sie eine `public`-Klasse `PairwiseDoubleArrayBinaryOperatorGivingArray`, die das `public`-Interface `DoubleAr- rayBinaryOperatorGivingArray` implementiert. Ein Objekt dieser Klasse hat ein `private`-Objektattribut vom Typ `Double- BinaryOperator`. Der `public`-Konstruktor hat einen Parameter vom selben formalen Typ und initialisiert das Attribut damit wie üblich.

Falls einer der beiden aktualen Parameterwerte der Methode `applyAsDoubleArray` gleich `null` ist, liefert diese Methode `null` zurück. Andernfalls liefert sie ein Array zurück, das genauso lang ist wie das kürzere ihrer beiden aktualen Parameter. An jeden Index i des zurückgelieferten Arrays steht das Ergebnis der Anwendung des Attributs auf den Wert an Index i in den beiden aktualen Parametern, wobei die Reihenfolge dieselbe ist: Der erste aktuale Parameter von `applyAsDouble` ist aus dem ersten aktualen Parameter von `applyAsDoubleArray` entnommen, der zweite entsprechend aus dem zweiten.

Dieser binäre Operator auf Arrays ist offensichtlich genau dann kommutativ, wenn der verwendete `DoubleBinaryOperator` kommutativ ist.

H3.4: Binäre Fold-Klasse auf „Array von `double`“ -1 Punkte

Schreiben Sie eine `public`-Klasse `PairwiseDoubleArrayBinaryOperatorGivingScalar`, die das `public`-Interface `DoubleAr- rayBinaryOperatorGivingScalar` implementiert. Ein Objekt dieser Klasse hat zwei `private`-Objektattribute vom Typ `DoubleBinaryOperator` sowie ein `private`-Objektattribut vom Typ `double`. Der `public`-Konstruktor hat drei Parameter derselben formalen Typen und initialisiert die drei Attribute damit wie üblich.

Der erste der beiden binären Operatoren wird im Folgenden die ***Komponentenverknüpfung***, der zweite die ***Faltungsoperation*** genannt.

Die Methode `applyAsDoubleArray` von `PairwiseDoubleArrayBinaryOperatorGivingScalar` übersetzt folgende Logik aus Racket in Java, wobei `join-fct` die Komponentenverknüpfung, `fold-fct` die Faltungsoperation und `init` das `double`-Attribut von `PairwiseDoubleArrayBinaryOperatorGivingScalar` ist. Der Durchgang von vorne nach hinten durch die Liste soll dabei übersetzt werden in aufsteigenden Durchlauf durch die Arrayindizes:

```
1 ( define ( apply lst1 lst2 join-fct fold-fct init )
2     ( cond
3        [  ( or ( empty? lst1 ) ( empty? lst2 ) )  init  ]
4        [  else  ( fold-fct
5                   ( join-fct ( first lst1 ) ( first lst2 ) )
6                   ( apply ( rest lst1 ) ( rest lst2 ) join-fct fold-fct init )
↪  ]  ) )
```

***Verbindliche Anforderung*:** Die Methode `applyAsDoubleArray` von `PairwiseDoubleArrayBinaryOperatorGivingScalar` wird durch eine einzige Schleife realisiert, das heißt, Rekursion ist nicht erlaubt und mehr als eine Schleife ist ebenfalls nicht erlaubt.

***Verständnisfragen am Rande* (0 Punkte):**

- Dieses Auswertungsschema ist linksassoziativ auf den beiden Listen in Racket bzw. Arrays in Java. Was bedeutet dies und wie würde man Rechtsassoziativität in Java erreichen? [2]
- Im Aufruf von `fold-fct` finden Sie wieder `fold-fct`. Ist das jetzt Rekursion oder was ist das sonst?

H4: Verwendung/Test der Klassen mit Hilfe von Lambda-Ausdrücken -1 Punkte

Richten Sie eine Variable vom statischen Typ `ReduceDoubleArray` ein und verwenden Sie als `DoublePredicate` einen Lambda-Ausdruck in Kurzform gemäß Folie 89-97 von Kapitel 04c. Dieser Lambda-Ausdruck testet einfach nur, ob der aktuale Parameterwert im Intervall $[100 \ldots 1000]$ ist.

Schreiben Sie eine `public`-Klasse `MyTinyMath` mit zwei `private`-Objektattributen vom Typ `boolean`, die wie üblich durch zwei Parameter des `public`-Konstruktors vom formalen Typ `boolean` initialisiert werden. Die `public`-Methode `conditio- nalSum` gibt die Summe aus den beiden aktualen Parameterwerten zurück, allerdings werden die aktualen Parameterwerte nach folgender Vorschrift teilweise verdoppelt: Keiner der beiden aktualen Parameterwerte wird verdoppelt, falls beide booleschen Attribute Wert `false` haben; nur der erste aktualen Parameterwert wird verdoppelt, falls nur das erste boolesche Attribut Wert `true` hat; nur der zweite aktualen Parameterwert, falls nur das zweite boolesche Attribut Wert `true` hat; schließlich beide aktuale Parameterwerte, falls beide boolesche Attribute Wert `true` haben. Diese Logik setzen Sie aber nicht 1:1 wie formuliert um, sondern mit nur zwei `if` ohne `else` (auch kein `switch` und kein Bedingungsoperator). Anders gesagt: Sie richten in `conditionalSum` eine `double`-Variable ein, in der sie sukzessive das Ergebnis berechnen.

Richten Sie vier Variable vom statischen Typ `MyTinyMath` und vier Objekte von `MyTinyMath` ein und lassen Sie jede Variable auf eines dieser Objekte verweisen (auf jedes Objekt verweist genau eine Variable). Die vier Objekte initialisieren Sie mit den vier verschiedenen möglichen Kombinationen von `true` und `false`.

Richten Sie als nächstes vier Variable vom statischen Typ `DoubleBinaryOperator` ein und initialisieren Sie jede dieser vier Variablen mit einer Methodenreferenz auf Methode `conditionalSum` von `MyTinyMath` gemäß Folien 186 ff. von Kapitel 04c. Da diese Methode eine Objektmethode ist, muss die Methodenreferenz noch den Namen einer Variable oder Konstante von `MyTinyMath` haben; dafür nehmen Sie die oben eingerichteten vier Variablen von `MyTinyMath`.

Richten Sie nun zwei Variable vom statischen Typ `PairwiseDoubleArrayBinaryOperatorGivingArray` ein und verwenden Sie als `DoubleBinaryOperator` zwei verschiedene der vier oben definierten Variablen vom statischen Typ `DoubleBinaryOpe- rator`, und zwar die beiden, bei denen die beiden booleschen Attribute verschiedene Werte haben (also mit einem Wert `true` und dem anderen Wert `false`).

Richten Sie eine Variable vom statischen Typ `PairwiseDoubleArrayBinaryOperatorGivingScalar` ein. Dafür verwenden Sie die anderen beiden oben definierten Variablen vom statischen Typ `DoubleBinaryOperator`, und zwar diejenige Variable, bei der beide booleschen Attribute Wert `true` haben, als Komponentenverknüpfung, die andere als Faltungsoperation.

Richten Sie acht Arrays von `double` ein, jeweils mit Länge 300. Für $i \in \{1, 2, 3, 4\}$ hat das i-te Array den Wert $i \cdot j$ an Position $j \in \{0, \ldots, 299\}$ und das $(4 + i)$-te Array den Wert $1200 - i \cdot j$. Wenden Sie die oben definierte Variable vom statischen Typ `ReduceDoubleArray` auf das erste, zweite, siebte und achte Array an. Prüfen Sie zur eigenen Sicherheit, ob diese Anwendung in jedem Fall das korrekte Ergebnis geliefert hat.

[2] Siehe bspw. `https://de.wikipedia.org/wiki/Operatorassoziativit%C3%A4t#Linksassoziative_Operatoren`.

Sie haben nun insgesamt zwölf Arrays, acht ursprüngliche und vier Reduktionen von ursprünglichen Arrays.

Wenden Sie jeden der vier oben auf Basis von `MyTinyMath` definierten `DoubleBinaryOperator` auf insgesamt 32 Paaren von Arrays an: Bei den ersten 16 Paaren ist das erste Array eines der urprünglichen Arrays Nr. 3, 4, 5 und 6, das erste Array ist die Reduktion eines der Arrays Nr. 1, 2, 3 und 4. Für $i \in \{1, \ldots, 16\}$ besteht das Paar $16 + i$ aus denselben beiden Arrays wie das Paar i, nur in umgekehrter Reihenfolge. Prüfen Sie auch hier zur eigenen Sicherheit, ob diese Anwendung in jedem Fall das korrekte Ergebnis geliefert hat.

Auf dieselben 32 Paare von Arrays wenden Sie beide oben definierten Variablen vom statischen Typ `PairwiseDoubleArray- BinaryOperatorGivingArray` sowie die oben definierte Variable vom statischen Typ vom statischen Typ `PairwiseDoubleAr- rayBinaryOperatorGivingScalar` an. Prüfen Sie auch hier zur eigenen Sicherheit, ob diese Anwendung in jedem Fall das korrekte Ergebnis geliefert hat.