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: 34 Punkte

Lambda-Ausdrücke

Beachten Sie die Seite *Verbindliche Anforderungen für alle Abgaben* im Moodle-Kurs.

Verstöße gegen verbindliche Anforderungen führen zu Punktabzügen und können die korrekte Bewertung Ihrer Abgabe beeinflussen. Sofern vorhanden, müssen die in der Vorlage mit `TODO` markierten `crash`-Aufrufe entfernt werden. Andernfalls wird die jeweilige Aufgabe nicht bewertet.

Die für diese Hausübung relevanten Verzeichnisse sind `src/main/java/h07` und ggf. `src/test/java/h07`.

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“, welches beides miteinander verbindet: ***Lambda-Ausdrücke***. Das ist eine der vielen schönen und nützlichen programmiersprachlichen Konstrukte, die in funktionalen Sprachen (wie Racket) entwickelt und in anderen Programmiersprachen (wie Java) übernommen worden sind. Leider leidet die Schönheit bei einer solchen Übertragung zwangsläufig, ist vielleicht aber immer noch sichtbar. Die Nützlichkeit ist jedenfalls, wie Sie sehen werden, auch in Programmiersprachen wie Java hoch.

Um Sie in dieser Hausübung also diesem, vielleicht anfangs etwas überfordernd wirkendem, Thema anzunähern, werden Sie den direkten Vergleich zwischen standardmäßigen Implementieren von Interfaces zur Nutzung von Lambda-Ausdrücke für Selbiges direkt sehen.

In der ersten Aufgabe (H1) werden Sie dabei zunächst einige Klassen, die ein Interface implementieren, wie gewohnt vervollständigen, indem Sie die übergebene Methode des Interfaces selbst implementieren. Die zweite Aufgabe (H2) verläuft dabei völlig analog: Auch hier vervollständigen Sie Klassen, die ein Interface implementieren.

Erst in der dritten Aufgabe (H3) erhalten Sie den ersten konkreten Kontakt mit Lambda-Ausdrücken. Hier werden Sie die Funktionalitäten der Methoden aus H2 nicht in standardmäßiger Form, sondern eben mittels Lambda-Ausdrücken implementiert. Dabei haben Sie dann einen direkten Vergleich zwischen der Ihnen bereits bekannten Art und Weise und dieser nun neu eingeführten, praktischeren Methode.

In der letzten Aufgabe (H4) vervollständigen Sie dann die letzten Reste einer Fabrik, die Ihnen, anhand der ihr übergebenen Spezifikationen, einen Operator zurückliefert.

H1: Unäre/Binäre Operatoren auf „Array von `double`“ als Funktionale Interfaces 11 Punkte

Hinweis:

Alle in *dieser Aufgabe* relevanten Klassen und Methoden befinden sich im Package `arrayoperators`.

Sie finden in der Vorlage zunächst drei Interfaces:

- Ein Interface namens `DoubleArrayUnaryOperatorGivingArray` mit einer funktionalen Methode `applyAsDoubleArray`, die einen Parameter vom formalen Typ „Array von `double`“ und Rückgabetyp „Array von `double`“ hat. Die Hauptfunktionalität von implementierenden Klassen soll es sein, eine einfache *Filteroperation* auf dem gegebenen Array zu implementieren. Dabei werden die Elemente, die den Filter „passieren“ in einem neuen Array gespeichert und dieses zurückgeliefert.

- Ein Interface namens `DoubleArrayBinaryOperatorGivingArray` mit der selben funktionalen Methode `applyAsDoubleArray`, die allerdings zwei Parameter vom formalen Typ „Array von `double`“ aufweist und ebenfalls Rückgabetyp „Array von `double`“ hat. Ziel einer Implementation der Methode ist es, dass eine gewisse *binäre Operation* auf beiden aktualen Parameterwerten realisiert wird. Das Resultat dieser Operation wird dann zurückgeliefert.

- Zuletzt noch ein Interface namens `DoubleArrayBinaryOperatorGivingDouble`, abermals mit der funktionalen Methode `applyAsDoubleArray`. Diese hat einen formalen Parameter vom Typ „Array von `double`“ und hat als Rückgabetyp `double`. Implementierende Klassen sollten hierbei mittels `applyAsDoubleArray` eine *Faltungsoperation* auf einem Array implementieren, die lediglich einen Wert zurückliefert.

Verbindliche Anforderung:

Jedes der Methode „`applyAsDoubleArray`“ (also einer beliebigen Implementation) übergebene Array darf *nicht* modifiziert werden.

Unbewertete Verständnisfrage:

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?

H1.1: Unäre Filter-Klasse auf „Array von `double`“ 4 Punkte

Passend zum ersten `DoubleArrayUnaryOperatorGivingArray` aus H1 finden Sie in der Codevorlage die Klasse `ReduceDoubleArray`, die das Interface implementiert. Ein Objekt dieser Klasse hat darüber hinaus eine Objektkonstante „`PREDICATE`“ vom Typ `DoublePredicate`[1].

Nun implementieren Sie die geerbte Methode `applyAsDoubleArray` wie folgt: Falls der aktuale Parameterwert der Methode `applyAsDoubleArray` gleich `null` ist, liefert diese Methode `null` zurück. Andernfalls liefert die Methode ein Array zurück, das höchstens so lang ist wie der aktuale Parameter (die Länge der Rückgabe kann auch 0 sein(!)). Dabei enthält die Rückgabe alle Komponenten, für die die Methode `test` des Prädikats `PREDICATE` `true` liefert. Diese Komponenten sind in der Rückgabe in derselben Reihenfolge, wie auch schon im aktualen Parameter. Darüber hinaus hat die Rückgabe keine weiteren Komponenten.

Dabei geht Ihre Methode `applyAsDoubleArray` 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.

[1] Gemeint ist natürlich `java.util.function.DoublePredicate`.

Anmerkung:

Hierbei empfiehlt es sich die Zählung der Komponenten der Rückgabe in einer separaten Methode zu zählen (analog zu Aufgabe H1.1 und H1.2 in der Hausübung H02), ist aber nicht zwangsläufig notwendig und wird auch nicht verlangt.

H1.2: Binäre Map-Klasse auf „Array von `double`“ 4 Punkte

Des Weiteren finden Sie in der Codevorlage die Klasse `PairwiseDoubleArrayBinaryOperatorGivingArray`, passend zum Interface `DoubleArrayBinaryOperatorGivingArray`. Ein Objekt dieser Klasse hat eine Objektkonstante vom Typ `DoubleBinaryOperator`[2] namens „`OPERATOR`“.

Falls einer der beiden aktualen Parameterwerte der Methode gleich `null` ist, liefert diese Methode `null` zurück. Sollte dies nicht der Fall sein, verläuft die Implementation der Methode `applyAsDoubleArray` nun anders als in der vorhergehenden Aufgabe.

Konkret liefert die Methode ein Array zurück, das genauso lang ist, wie das *kürzere* der beiden aktualen Parameter. Das Array wird nun wie folgt befüllt: An jedem Index i des zurückgelieferten Arrays steht das Ergebnis der Anwendung des Attributs `operator` auf den Wert am selben Index i in den beiden aktualen Parametern. Hierbei ist die Reihenfolge dieselbe. Der erste aktuale Parameter von `applyAsDouble` ist aus dem ersten aktualen Parameter von `applyAsDoubleArray` entnommen, der zweite entsprechend aus dem zweiten.

Unbewertete Verständnisfrage:

Ist dieser binäre Operator auf Arrays kommutativ? Wenn ja, wann?

H1.3: Binäre Fold-Klasse auf „Array von `double`“ 3 Punkte

Als letzte Klasse dieser Aufgabe finden Sie `PairwiseDoubleArrayBinaryOperatorGivingScalar`, die zum vorher erwähnten Interface `DoubleArrayBinaryOperatorGivingScalar` passt. Ein Objekt dieser Klasse hat zwei Objektkonstanten namens „`OPERATOR_1`“ und „`OPERATOR_2`“ vom Typ `DoubleBinaryOperator` sowie ein Objektattribut `init` vom Typ `double`. Im Folgenden bezeichnet der **erste** binäre Operator `OPERATOR_1` die ***Komponentenverknüpfung*** und der **zweite** binäre Operator `OPERATOR_2` die ***Faltungsoperation***.

Im Gegensatz zu den vorhergehenden Methoden dürfen Sie bei Ihrer Implementation davon ausgehen, dass die übergebenen Parameter nie auf `null` verweisen. Ebenfalls anders als in den beiden vorhergehenden Aufgaben übersetzt die Implementation der Methode `applyAsDoubleArray` die folgende Logik aus der Programmiersprache Racket in Java:

```
( define ( apply lst1 lst2 join-fct fold-fct init )
( cond
    [  ( or ( empty? lst1 ) ( empty? lst2 ) )  init  ]
    [  else  ( fold-fct
        ( join-fct ( first lst1 ) ( first lst2 ) )
        ( apply ( rest lst1 ) ( rest lst2 ) join-fct fold-fct init ) ) ] ) )
```

Hinweis:

Hierbei versteht sich `join-fct` als die *Komponentenverknüpfung*, `fold-fct` als die *Faltungsoperation* und `init` als das `double`-Attribut von `PairwiseDoubleArrayBinaryOperatorGivingScalar`

[2] Gemeint ist natürlich `java.util.function.DoubleBinaryOperator`

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.

Unbewertete Verständnisfragen:

- 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? (Siehe bspw. `https://de.wikipedia.org/wiki/Operatorassoziativit%C3%A4t#Linksassoziative_Operatoren`.)
- Im Aufruf von `fold-fct` finden Sie wieder `fold-fct`. Ist das jetzt Rekursion oder was ist das sonst?
- Über diese Aufgabe hinaus, finden Sie die oben (H1) am Anfang der Aufgabe beschriebenen Anforderungen an die implementierenden Klassen bezüglich der Operationen erfüllt? Fallen Ihnen Beispiele für die einzelnen Operationen ein?

H2: Binäre Operatoren auf `double` als Funktionale Interfaces — 4 Punkte

Hinweis:

Alle in dieser Aufgabe relevanten Klassen befinden sich im Packge `doubleoperators`.

H2.1: Erste binäre Operatorklasse auf `double` — 1 Punkt

In der Codevorlage finden Sie nun eine Klasse `DoubleSumWithCoefficientsOp`[3], die das Interface `DoubleBinaryOperator` implementiert.

Klasse `DoubleSumWithCoefficientsOp` besitzt zwei Attribute, `COEFF_1` und `COEFF_2`, vom primitiven Datentyp `double`. Darüber hinaus erhält die Methode `applyAsDouble` ebenfalls zwei `double` Werte als Parameter, nämlich `left` als den Ersten und `right` als den Zweiten.

Konkret soll Ihre Methode `applyAsDouble` ihren ersten Parameter `left` mit `COEFF_1` und ihren zweiten Parameter `right` mit `COEFF_2` multiplizieren und die Summe dieser beiden Produkte zurückliefern.

Unbewertete Verständnisfrage:

Ist dieser binäre Operator kommutativ? Was ist im Fall `COEFF_1 == COEFF_2`?

H2.2: Zweite binäre Operatorklasse auf `double` — 1 Punkt

Sie finden in der Vorlage noch die Klasse `EuclideanNorm`, die das Interface `DoubleBinaryOperator` implementiert.

`applyAsDouble` implementieren Sie hier so, dass sie für ihre aktualen Parameterwerte `x` und `y` vom Typ `double` den eindimensionalen euklidischen Abstand[4], also konkret den Wert $\sqrt{x^2 + y^2}$, mit Hilfe der Klassenmethode `sqrt` von

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

Klasse `java.lang.Math` zurückliefert.

H2.3: Dritte binäre Operatorklasse auf `double` — 1 Punkt

Als dritte Operatorklasse finden Sie noch `DoubleMaxOfTwo`, die abermals das Interface `DoubleBinaryOperator` implementiert. Diese liefert in ihrer Methode `applyAsDouble`, wie der Name der Klasse bereits vermuten lässt, das Maximum ihrer beiden aktualen Parameterwerte, `left` und `right` zurück.

H2.4: Vierte binäre Operatorklasse auf `double` — 1 Punkt

Als letztes implementieren Sie noch die Methode `applyAsDouble` in Klasse `ComposedDoubleBinaryOperator`. Ein Objekt dieser Klasse besitzt selbst drei Objektkonstanten vom Typen `DoubleBinaryOperator`.

Konkret ist die Funktionalität ihrer `applyAsDouble` Methode nun wie folgt: Zunächst wird der erste Operator `OP_1` die beiden aktualen Parameterwerte `left` und `right` angewandt und separat davon auch noch der zweite Operator `OP_2` auf die selben übergebenen `double`-Werte (in der selben Reihenfolge). Schließlich wird nun der dritte Operator `OP_3` auf das Ergebnis dieser beiden Operatoren angewandt und das Resultat von der Methode zurückgeliefert.

> **Unbewertete Verständnisfrage:**
>
> Was gilt bezüglich Kommutativität bei diesem Operator?

H3: Lambda-Ausdrücke in Kurzform und Standardform — 10 Punkte

> **Hinweise:**
>
> - Auch die für diese Aufgabe relevante Klasse, `DoubleBinaryOperatorFactory` befindet sich einfach im Package `h07`. Darüber hinaus befinden sich im Package `doubleoperators` zwei Klassen, `PairOfDoubleCoefficients` und `TripleOfDoubleBinaryOperators`. Die Klasse `PairOfDoubleCoefficients` weist zwei `public`-Objektattribute „`coeff1`“ und „`coeff2`“ vom Typ `double` auf. `TripleOfDoubleBinaryOperators` besitzt drei `public`-Objektattribute „`op1`“, „`op2`“ und „`op3`“ vom Typ `DoubleBinaryOperator`. Diese werden in der folgenden Aufgabe benötigt.
> - Im Folgenden bezieht sich die ***Standard-*** und die ***Kurzform*** eines Lambda-Ausdruckes auf die Darstellungen gemäß Kapitel 04c, Folien 89-97 der FOP

> **Verbindliche Anforderung:**
>
> Alle Lambda-Ausdrücke, die Sie in dieser Aufgabe schreiben, dürfen nicht einfach ein Objekt der zu ersetzenden Klasse liefern. Heißt konkret: Die Verwendung von `new` ist in allen Aufgaben der H3 untersagt.

H3.1: Lambda-Ausdruck anstelle von `DoubleSumWithCoefficientsOp` — 2 Punkte

In der Klasse `DoubleBinaryOperatorFactory` finden Sie die Methode `doubleSumWithCoefficientsOpAsLambda`. Diese erhält einen aktualen Parameter vom Typ `Object` und liefert ein `DoubleBinaryOperator`-Objekt zurück.

Die Methode implementieren Sie wie folgt: Sollte der dynamische Typ des aktualen Parameters `PairOfDoubleCoefficients` *oder* ein Subtyp davon sein, erstellen Sie die Rückgabe mittels eines Lambda-Ausdrucks in ***Standardform***. Dabei ist die Logik dieser Rückgabe äquivalent zu der Funktion der Methode `applyAsDouble` von `DoubleSumWithCoefficientsOp`. Die Koeffizienten entnehmen Sie dabei 1 zu 1 aus dem `PairOfDoubleCoefficients`-Objekt im aktualen Parameter.

Andernfalls wird `null` von `doubleSumWithCoefficientsOpAsLambda` zurückgeliefert.

H3.2: Lambda-Ausdruck anstelle von `EuclideanNorm` — 2 Punkte

Sie finden außerdem eine Methode namens `euclideanNormAsLambda` in der Klasse `DoubleBinaryOperatorFactory`. Die Rückgabe dieser Methode erstellen Sie wieder mittels eines Lambda-Ausdrucks in ***Standardform***. Dabei ist die Logik dieses Ausdrucks äquivalent zu der Funktion der Methode `applyAsDouble` in `EuclideanNorm`.

H3.3: Lambda-Ausdruck anstelle von `DoubleMaxOfTwo` — 4 Punkte

Als nächstes implementieren Sie die Methode `doubleMaxOfTwoAsLambda`. Diese erhält wie auch schon die Methode `doubleSumWithCoefficientsOpAsLambda` einen aktualen Parameter vom Typ `Object` und auch hier entscheidet sich die Art der Rückgabe anhand dieses Objektes.

Ist der dynamische Typ des aktualen Parameters gleich `Boolean`, wird anhand des in diesem Objekt eingekapselten bool'schen Wertes die Rückgabe bestimmt: Kapselt das Objekt `true` ein, wird hier der Lambda-Audruck **nicht** in ***Standardform***, sondern in ***Kurzform*** gebildet, wobei hier die Maximumsberechnung keine Methode verwendet, sondern mit Hilfe des Bedingungsoperators „<“ bestimmt wird.

Kapselt das `Boolean`-Objekt hingegen `false` ein, soll als Lambda-Ausdruck eine Methodenreferenz mit der Methode `max` von Klasse `Math` verwendet werden (siehe dazu Kapitel 04c, Folien 186-213 der FOP).

Ist der dynamische Typ des aktualen Parameters **nicht** `Boolean`, liefert diese Methode `null` zurück.

> **Hinweis:**
>
> Sollte Ihnen Ihre IDE den Hinweis geben, dass sie Ihren Bedingungsoperator „<“ durch einen Aufruf von `Math.max()` ersetzen können, ignorieren Sie diesen.

H3.4: Lambda-Audruck anstelle von `ComposedDoubleBinaryOperator` — 2 Punkte

Zuletzt fehlt noch die Methode `composedDoubleBinaryOperatorAsLambda`. Auch sie erhält einen aktualen Parameter vom Typ `Object` und auch hier entscheidet sich die Art der Rückgabe anhand dieses Objektes.

Sollte der dynamische Typ des aktualen Parameters `TripleOfDoubleBinaryOperators` oder ein Subtyp davon sein, erstellen Sie die Rückgabe mittels eines Lambda-Ausdrucks in ***Kurzform***. Dabei ist die Logik dieser Rückgabe äquivalent zu der Funktion der Methode `applyAsDouble` von `ComposedDoubleBinaryOperator`. Die Koeffizienten entnehmen Sie dabei 1 zu 1 aus dem `TripleOfDoubleBinaryOperators`-Objekt im aktualen Parameter.

Andernfalls wird `null` von `composedDoubleBinaryOperatorAsLambda` zurückgeliefert.

Anhand dieser Aufgabe haben Sie die eigentliche Stärke von Lambda-Ausdrücken kennen gelernt. Um bisher beispielsweise die relativ simple Funktionalität der oben eingeführten Operatoren zu implementieren, brauchten wir bisher eine

Klasse, die eine Methode (hier `applyAsDouble`) eines Interfaces (hier `DoubleBinaryOperator`) implementiert. Wie Sie jetzt in dieser Aufgabe gelernt haben, geht dies auch wesentlich eleganter von statten, ohne große Implementationen oder ohne lästiges Erstellen von Objekten, nämlich einfach mit einem einzigen Lambda-Ausdruck in Standard- oder Kurzform. Diese wundervolle Funktionalität der Programmiersprache Java wird Ihnen immer wieder über den Weg laufen, beispielsweise beim Umgang mit Listen (Stichwort Sortierung, Filterung, etc.).

H4: Das Bauen von Operatoren mit Hilfe der Klasse `DoubleBinaryOperatorFactory` 9 Punkte

Anhand der vorhergehenden Aufgabe haben Sie die eigentliche Stärke von Lambda-Ausdrücken kennen gelernt. Um bisher beispielsweise die relativ simple Funktionalität der oben eingeführten Operatoren zu implementieren, brauchten wir bisher eine Klasse, die eine Methode (hier `applyAsDouble`) eines Interfaces (hier `DoubleBinaryOperator`) implementiert. Wie Sie jetzt in dieser Aufgabe gelernt haben, geht dies auch wesentlich eleganter von statten, ohne große Implementationen oder ohne lästiges Erstellen von Objekten, nämlich einfach mit einem einzigen Lambda-Ausdruck in Standard- oder Kurzform. Diese wundervolle Funktionalität der Programmiersprache Java wird Ihnen immer wieder über den Weg laufen, beispielsweise beim Umgang mit Listen (Stichwort Sortierung, Filterung, etc.).

In dieser Aufgabe nun soll dieser Vorteil nun noch einmal zur Geltung kommen, denn hier erstellen Sie die eigentliche Fabrik, die Ihnen anhand gewisser Spezifikationen Operatoren erstellt und zurückgeliefert. Dabei werden einmal die von Ihnen in Aufgabe H3 implementierten Klassen mittels `new` erstellt und auf der anderen Seite einfach die soeben implementierten Methoden aufgerufen.

H4.1: Die Methode `buildOperator` 1 Punkt

Sie finden also in der Klasse `DoubleBinaryOperatorFactory` noch eine weitere Methode namens `buildOperator`. Diese erwartet mehrere Spezifikationen als aktuale Parameter. Der erste aktuale Parameter vom Typ `String` legt die Art des `DoubleBinaryOperator`-Objektes fest, die die Fabrik zurückliefert. Der zweite aktuale Parameter vom Typ `Object`[5] legt, unter Umständen, eine bestimmte Initialisierung des zurückgelieferten `DoubleBinaryOperator`-Objektes fest und der dritte, bool'sche, aktuale Parameter entscheidet über die Art und Weise, wie der Operator erstellt wird.

Die Rückgabe der Methode ist `null`, sollte der erste aktuale Parameter keiner der vier Strings `"Coeffs"`, `"Euclidean"`, `"Max"` oder `"Composed"` beinhaltet.

Sollte der dritte Parameter nun „`true`“ eingekapseln, ruft `buildOperator` die Methode `buildOperatorWithNew` auf. Dabei werden die ersten beiden aktualen Parameter von `buildOperator` einfach für `buildOperatorWithNew` übernommen.

Ist der Wert des dritten Parameters allerdings `false`, wird stattdessen die Methode `buildOperatorWithLambda` aufgerufen. Auch diese erhält die selben Parameter, wie auch schon `buildOperatorWithNew`.

H4.2: Operatoren mittels `new` 3 Punkte

Nun implementieren Sie zunächst die Methode `buildOperatorWithNew`. Wir unterscheiden die Art des zurückgelieferten Operators dabei wie folgt anhand des ersten aktualen Parameters `operator`:

- `"Coeffs"`: Nun entscheidet der zweite aktuale Parameter über die Rückgabe. Ist der dynamische Typ des Parameters `PairOfDoubleCoefficients` oder ein Subtyp davon ist, ist der dynamische Typ der Rückgabe

[5] Warum wir diesen Typen absichtlich so flexibel gehalten haben, erfahren Sie gleich.

`DoubleSumWithCoefficientsOp` und die zwei Koeffizienten entnehmen Sie aus dem zweiten aktualen Parameter. Andernfalls liefert `buildOperator` `null` zurück.

- `"Euclidean"`: Hier wird der zweite aktuale Parameter der Methode ignoriert. Der dynamische Typ der Rückgabe ist einfach `EuclideanNorm`.
- `"Max"`: Auch hier wird der zweite aktuale Parameter der Methode ignoriert. Der dynamische Typ der Rückgabe ist einfach `DoubleMaxOfTwo`.
- `"Composed"`: Hier entscheidet der zweite aktuale Parameter wieder. 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, wie bereits im ersten Fall, aus dem zweiten aktualen Parameter geholt. Andernfalls wird wieder `null` von `buildOperator` zurückgeliefert.

Verbindliche Anforderung:

Die Fallunterscheidung anhand des ersten Parameters soll in einer einzigen `switch`-Anweisung (siehe Kapitel 03c, Folien 214-226 der FOP) geschehen.

H4.3: Operatoren mittels Lambda-Ausdrücken — 5 Punkte

Die Methode `buildOperatorWithLambda` funktioniert nun ziemlich analog zur Methode `buildOperatorWithNew`. Auch hier machen Sie eine Fallunterscheidung anhand des ersten Parameters `operator`. Allerdings wird hier nun die Rückgabe der Methode in jedem der Fälle durch einen Aufruf der in Aufgabe H3 implementierten Methoden gewährleistet:

- `"Coeffs"`: Sie liefern die Rückgabe des Aufrufs der Methode `doubleSumWithCoefficientsOpAsLambda` zurück. Dabei übergeben Sie der Methode den zweiten aktualen Parameter der Methode `buildOperatorWithLambda`.
- `"Euclidean"`: Sie liefern die Rückgabe des Aufrufs der Methode `euclideanNormAsLambda` zurück.
- `"Max"`: Sie liefern die Rückgabe des Aufrufs der Methode `doubleMaxOfTwoAsLambda` zurück. Dabei übergeben Sie der Methode erneut den zweiten aktualen Parameter der Methode `buildOperatorWithLambda`.
- `"Composed"`: Sie liefern die Rückgabe des Aufrufs der Methode `composedDoubleBinaryOperatorAsLambda` zurück. Dabei übergeben Sie, völlig analog zum ersten Fall, der Methode den zweiten aktualen Parameter der Methode `buildOperatorWithLambda`.

Verbindliche Anforderungen:

- In dieser Aufgabe dürfen keine Objekte mittels des Operators `new` erstellt werden.
- Realisieren Sie die Methode mittels eines einzigen `return`-Statements. Dabei ist es Ihnen auch nicht erlaubt die eigentliche Funktionalität in eine weitere Methode auszulagern.

Unbewertete Verständnisfragen:

- Sehen Sie hier irgendwo Closure gemäß Kapitel 04c, Folien 68-70 der FOP?
- 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?