Funktionale und objektorientierte Programmierkonzepte
Übungsblatt 09

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 09 — Gesamt:

Ein Einblick in Generics

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/h09` und ggf. `src/test/java/h09`.

Einleitung

In dieser Hausübung beschäftigen wir uns überwiegend mit dem Thema Generizität. Mit Generics werden im Java-Umfeld Sprachmittel bezeichnet, mit denen Klassen und Methoden mit Typparametern parametrisiert werden können, um Typsicherheit trotz Typunabhängigkeit zu ermöglichen.

In den folgenden Aufgaben werden Sie aus praktischer Perspektive sehen, welche Vorteile sich durch die Nutzung von Generics ergeben. Mit Generics demonstrieren wir in der letzten Aufgabe dann auch die Funktionsweise und den Sinn von JUnit-Tests.

H1: Hilfsklassen

H1.1: Factories für `Number`-Klassen -1 Punkte

In Kapitel 06 haben Sie gelesen, dass Generics in Java leider ein paar Einschränkungen haben, die aus der historischen Entwicklung von Java entstanden sind. Dazu gehört, dass keine Objekte von einem generischen Typparameter mit `new` erzeugt werden dürfen. Dummerweise will man so etwas aber häufiger. Diese Einschränkung kann man mit einem Entwurfsmuster namens *Fabrikmethode*[1] (*factory method*) umgehen, das sehr viel allgemeiner verwendbar ist und mit dem die Erzeugung von Objekten variiert werden kann, ohne dass der Kontext der Erzeugung (in Java: die Methode, in der es erzeugt wird) dafür geändert werden muss.

Schreiben Sie also ein `public`-Interface `NumberFactory1`, das generisch mit einem generischen Typparameter `T`, der auf `Number` und alle Subtypen von `Number` eingeschränkt ist. Das Interface `NumberFactory1` hat eine Methode `get` mit einem Parameter vom formalen Typ `Double` und Rückgabetyp `T`.

Schreiben Sie eine nichtgenerische Klasse `DoubleFactory1`, die `NumberFactory1<Double>` implementiert, sowie eine nichtgenerische Klasse `NumberAsDoubleFactory1`, die `NumberFactory1<Number>` implementiert. Die Methode `get` liefert in beiden Klassen ein `Double`-Objekt zurück, das den aktualen Parameter einkapselt.

Schreiben Sie eine nichtgenerische Klasse `IntegerFactory1`, die `NumberFactory1<Integer>` implementiert. Die Methode `get` liefert ein `Integer`-Objekt zurück, das den aktualen Parameter einkapselt. Dafür ist die Klassenmethode `valueOf` von Klasse `Integer` zu verwenden.

Schreiben Sie völlig analog ein `public`-Interface `NumberFactory2` sowie `public`-Klassen `DoubleFactory2`, `NumberAsDoubleFactory2` und `IntegerFactory2`, nur mit einem Unterschied: Bei `*Factory1` waren die Klassen selbst generisch, bei `*Factory2` ist die Klasse selbst nicht generisch, sondern stattdessen ist nur die Methode `get` generisch mit Typparameter `T`.

H1.2: Traits für die binären Operatorenklassen -1 Punkte

Laut Wikipedia ist ein *Trait* „eine wiederverwendbare Sammlung von Methoden und Attributen, ähnlich einer Klasse“.[2] Die Klasse `java.lang.Math` ist ein Beispiel dafür, sie besteht nur aus universell verwendbaren mathematischen Konstanten und Funktionen. Wir definieren hier Traits, die nur Methoden enthalten. Damit diese Methoden austauschbar sind, implementieren alle Traits-Klassen dasselbe Interface `BasicBinaryOperations`.

Schreiben Sie also ein `public`-Interface `BasicBinaryOperations`, das generisch mit Typparametern `X` und `Y` ist. Dieses Interface hat drei Methoden namens `add`, `mult` und `isLessThan`.

Die Methode `add` hat zwei Parameter vom formalen Typ `X` und Rückgabetyp `X`. Die Methode `mult` hat einen ersten Parameter vom formalen Typ `Y`, einen zweiten Parameter vom formalen Typ `X` und Rückgabetyp `X`. Die Methode `isLessThan` ist boolesch und hat zwei Parameter vom formalen Typ `X`.

Schreiben Sie eine `public`-Klasse `StandardBasicBinaryOperations1`, die generisch mit einem Typparameter `T` ist, der auf `Number` und die Subtypen von `Number` beschränkt ist. Die Klasse `StandardBasicBinaryOperations1` implementiert das Interface `BasicBinaryOperations1`, wobei `T` sowohl die Rolle von `X` als auch die Rolle von `Y` einnimmt. Sie hat eine `private`-Objektkonstante vom Typ `NumberFactory1<T>` und einen `public`-Konstruktor mit einem Parameter desselben formalen Typs `NumberFactory1<T>`, der das Attribut wie üblich initialisiert. Die

[1] https://de.wikipedia.org/wiki/Fabrikmethode
[2] https://de.wikipedia.org/wiki/Trait_(Programmierung)

drei Methoden sind mittels der in Java eingebauten Operatoren für Addition, Multiplikation und Vergleich von Werten primitiver Zahlentypen implementiert. Die Rückgabe wird durch das Attribut konstruiert.

Schreiben Sie eine `public`-Klasse `StandardBasicBinaryOperations2`, die generisch mit einem Typparameter `T` ist, der auf `Number` und die Subtypen von `Number` beschränkt ist. Der einzige Unterschied zu `StandardBasicBinaryOperations1` ist, dass das Attribut nun vom Typ `NumberFactory2` ist.

Schreiben Sie eine `public`-Klasse `StringBasicBinaryOperations`, die das Interface `BasicBinaryOperations1` implementiert, wobei `X == String` und `Y == Integer` ist. Die Methode `add` liefert einfach eine Konkatenation ihrer beiden aktualen Parameter zurück (vgl. Abschnitt zu Strings in Kapitel 03b). Die Methode `mult` liefert einen leeren String (nicht `null`!) zurück, falls der eingekapselte Wert n des ersten aktualen Parameters nichtpositiv (also ≤ 0) ist, andernfalls die Konkatenation von n Kopien des zweiten aktualen Parameters. Die Methode `isLessThan` verwendet `compareTo` von `String`, wobei genau dann `true` zurückgeliefert wird, wenn das Objekt, mit dem `compareTo` aufgerufen wird, strikt dem aktualen Parameter von `compareTo` lexikographisch vorgeordnet ist.

> **Anmerkung:**
>
> Das Beispiel `StringBasicBinaryOperations` demonstriert einen häufigen Fall: Addition und Vergleich finden ganz innerhalb des Typs statt, bei der Multiplikation hat einer der beiden Faktoren einen anderen Typ, typischerweise einen Zahlentyp. Auch in der Mathematik ist diese Konstellation grundlegend.[a] Die beiden Interfaces `BasicBinaryOpe- rations*` modellieren diesen Fall in ihren beiden generischen Typparametern. Die beiden Klassen `StandardBasicBinary- Operations*` modellieren den einfacheren Fall, dass alles in einem einzigen Typ statfindet, also beide Typen identisch sind. Natürlich hätte man die generischen Typparameter von `BasicBinaryOperations*` noch wesentlich allgemeiner definieren können, aber da muss dann – wie so häufig – subjektiv abgewogen werden, ob dieser höhere Allgemeinheitsgrad ausreichend nützlich für weitere Anwendungsfälle ist oder ob nicht die kompliziertere Handhabung beim Programmieren zu sehr negativ ins Gewicht fällt.
>
> [a] `https://de.wikipedia.org/wiki/Skalarmultiplikation`

H2: Binäre Operatoren als generische Functional Interfaces

Die Klassen in dieser Aufgabe sollen im Package `h09.h1` erstellt werden. Satz ← 1:1 aus 21/22 übernommen: ändern?

In dieser Aufgabe implementieren Sie einige Klassen und Methoden in Java, die völlig analog zu den gleichnamigen Klassen und Methoden in Hausübung 07 sind. Der Unterschied ist, dass diese Klassen und Methoden nun ***generisch*** sind.

Mit `BinaryOperator` ist im Folgenden das generische Functional Interface `java.util.function.BinaryOperator<T>` gemeint. Für eine Variable oder Konstante `op` vom statischen Typ `BinaryOperator<T>` mit `op != null` sagen wir bei Aufruf `op.apply(`x, y`)` im Folgenden, dass `op` auf x und y ***angewendet*** wird (wobei x und y natürlich vom Typ `T` sind).

H2.1: Erster Satz von binären Operatorklassen (noch nicht generisch) -1 Punkte

Schreiben Sie die drei Klassen aus H1.1, H1.3 und H1.4 von Hausübung 07 nochmals, aber nun implementieren diese Klassen nicht mehr `DoubleBinaryOperator`, sondern `BinaryOperator<Double>`.

H2.2: Zweiter Satz von binären Operatorklassen (nun generisch eingeschränkt auf `Number`) -1 Punkte

Schreiben Sie die drei Klassen aus H2.1 oben nochmals, aber nun sind diese Klassen generisch und implementieren nicht mehr `BinaryOperator<Double>`, sondern stattdessen `BinaryOperator<T>`. Bei den ersten beiden Klassen ist der generische Typparameter `T` allerdings eingeschränkt auf Klasse `Number` (gemeint ist `java.lang.Number`) und alle Subtypen von `Number`. Bei der dritten Klasse (`Composed...`) ist der generische Typparameter hingegen uneingeschränkt.

Konsequenterweise ersetzen Sie den Namensbestandteil `Double` in den Namen der ersten beiden Klassen durch `Number` und streichen ihn bei der dritten Klasse ersatzlos.

> **Hinweis** (auch für spätere Teilaufgaben)**:**
>
> Schlagen Sie die Methode `doubleValue` von `Number` nach und verwenden Sie diese.

> ***Fehlermeldungen besser verstehen*:** Was passiert, wenn Sie den generischen Typparameter `T` bei einer dieser Klassen versuchsweise mit `String` oder `Object` instantiieren?

> **Unbewertete Verständnisfrage:**
>
> Warum werden die ersten beiden Klassen auf `Number` und alle Subtypen von `Number` eingeschränkt und die dritte Klasse nicht?

H2.3: Dritter Satz von binären Operatorenklassen (uneingeschränkt generisch mittels Traits) -1 Punkte

Schreiben Sie nun noch die ersten beiden Klassen aus H2.1 oben noch jeweils zweimal, aber nun uneingeschränkt generisch. Jede der ersten vier Klassen hat eine `private`-Objektkonstante vom Typ `BasicBinaryOperations1` sowie einen `public`-Konstruktor mit einem Parameter vom selben formalen Typ `BasicBinaryOperations1`, der das Attribut wie üblich initialisiert. Die Methode `apply` jeder dieser drei Klassen verwendet die Methoden dieses Attributs anstelle der entsprechenden arithmetischen Operationen. Die zweiten vier Klassen unterscheiden sich von den ersten vier nur dadurch, dass

Bei der dritten Klasse haben Sie schon in H2.2 den Namensbestandteil `Double` ersatzlos gestrichen. Das machen Sie jetzt auch bei diesen vier Klassen, hängen allerdings an den Namen eine `1` bei der Version mit `BasicBinaryOperations1` und eine `2` bei der mit `BasicBinaryOperations2`.

H3: In- und Out-Parameter mit Wildcards

Am Ende des Abschnitts zu Wildcards in Kapitel 06 finden Sie eine Liste von Empfehlungen, wann man die formalen Typen einzelner Parameter nach oben bzw. nach unten beschränken sollte und wann man besser überhaupt keine Wildcards verwenden sollte. Wir folgen diesen Empfehlungen.

H3.1: In-Parameter -1 Punkte

Schreiben Sie eine nichtgenerische `public`-Klasse `NumberAndSubFactory` mit einer `public`-Objektmethode `get`. Die Methode `get` hat einen Parameter vom formalen Typ `T`, der auf `Number` und alle Subtypen von `Number` beschränkt ist. Sie hat Rückgabetyp `Double` und liefert mittels `doubleValue` den vom aktualen Parameter eingekapselten Wert zurück.

H3.2: Out-Parameter -1 Punkte

Schreiben Sie eine generische `public`-Klasse `Holder` mit einem generischen Typparameter `T` und einer `public`-Objektvari- able `held` vom Typ `T`.

Schreiben Sie eine `public`-Klasse `IntegerAndSuperFactory`, die analog zu `IntegerFactory` definiert ist, allerdings mit folgenden Unterschieden: Die Klasse `IntegerAndSuperFactory` selbst ist nicht generisch (und die Methode `get` ebenfalls nicht). Die Methode `get` hat einen ersten Parameter vom formalen Typ `Holder<T>`, aber `T` ist darin auf `Integer` und alle Supertypen von `Integer` eingeschränkt ist. Der zweite Parameter ist vom formalen Typ `int`. Diese Methode `get` richtet ein `Integer`-Objekt ein, das den zweiten aktualen Parameterwert einkapselt, und lässt Attribut `held` des ersten aktualen Parameter darauf verweisen.

Völlig analog zu `IntegerAndSuperFactory` schreiben Sie eine Klasse `ShortAndSuperFactory`, in der einfach nur Klasse `Integer` durch Klasse `Short` ersetzt ist.

Unbewertete Verständnisfragen:

- Der primitive Datentyp `Short` lässt sich ja implizit in `int` konvertieren. Geht das auch von `Short` nach `Integer`? Oder können Sie `get` von `ShortAndSuperFactory` mit einem `Integer`-Objekt als aktualen Parameter aufrufen? Wie begründen Sie Ihre Antwort?
- Warum ist für `get` nicht einfach Rückgabe vom Typ `T` möglich? Warum nicht Rückgabe vom Typ `Integer`? Warum nicht einfach eine Record-Klasse (Kapitel 03b) mit einem Attribut vom Typ `Integer` anstelle von `Holder`?

H4: Kompiliertests

In dieser Teilaufgabe kommt es nur auf Kompilierbarkeit an. Schreiben Sie dazu eine `public`-Klasse `TestBinOpConcepts` mit einer Methode `test`, die für die Klassen und Methoden aus H1+2 oben jeweils durch Einrichtung von Variablen und Objekten dieser Klassen mit ausgewählten Instanziierungen der generischen Typparameter demonstriert, dass jede Klasse und jede Methode tatsächlich so allgemein instantiiert werden kann, wie in der Vorlesung behauptet.

Konkret instantiieren Sie alle generischen Klassen aus H1+2 jeweils mit `Number`, `Double` und `Integer`. Diejenigen generischen Klassen aus H1+2, die nicht auf `Number` und die Subtypen von `Number` beschränkt sind, instantiieren Sie zusätzlich einmal mit `String`.

Die Methode `get` aus H3.1 instantiieren Sie ebenfalls mit `Number`, `Double` und `Integer`, die Methode `get` aus H3.2 mit `Integer`, `Number`, `Object` und `java.io.Serializable`.

Unbewertete Verständnisfrage:

warum `java.io.Serializable`?

H5: JUnit-Tests

H5.1: Zu testende Funktionalität -1 Punkte

Schreiben Sie eine generische `public`-Klasse `FilterCounter` mit uneingeschränktem generischem Typparameter `T`, die das Interface `Consumer` aus Package `java.util.function` implementiert. Diese Klasse hat eine `private`-Objektvariable vom Typ `int` sowie eine `private`-Objektkonstante vom Typ `Predicate<T>` (ebenfalls aus `java.util.function`). Sie hat einen `public`-Konstruktor mit einem Parameter vom formalen Typ `Predicate<T>`, mit dem das zweite Attribut wie üblich initialisiert wird. Das erste Attribut wird mit `0` initialisiert, muss also gar nicht explizit initialisiert werden (warum?). Die Klasse `FilterCounter` hat eine parameterlose `public`-Methode `getCounter`

Schreiben Sie eine generische `public`-Klasse `Accumulator` mit uneingeschränktem generischem Typparameter `T`, die ebenfalls `Consumer` implementiert. Diese Klasse hat eine `private`-Objektvariable `accu` vom Typ `T` und eine `private`-Objektkonstante vom Typ `BinaryOperator<T>`. Sie hat einen `public`-Konstruktor mit einem ersten Parameter vom formalen Typ `T` und einen zweiten Parameter vom formalen Typ `BinaryOperator<T>`, mit dem die beiden Attribute wie üblich initialisiert werden. Die Methode `accept` wendet das zweite Attribut auf das erste Attribut und den aktualen Parameter von `accept` an und überschreibt das erste Attribut mit dem Ergebnis dieser Anwendung. Bei dieser Anwendung muss unbedingt die Reihenfolge eingehalten werden: Das erste Attribut ist der erste aktuale Parameter von `apply`.

Unbewertete Verständnisfragen:

- Was hat `FilterCounter` mit der Standardoperation `filter` aus Kapitel 04c zu tun?
- Was hat `Accumulator` mit der Standardoperation `fold` aus Kapitel 04c zu tun?
- Warum ist die Reihenfolge bei der Anwendung des zweiten Attributs im `Accumulator` wichtig? Denken Sie an das Stichwort *Kommutativität*.

Noch (etwas!) mehr zu testende Funktionalität bzw. diese zu testende Funktionalität noch (etwas!) komplexer ausgestalten?

H5.2: JUnit-Tests -1 Punkte

Zu füllen!