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: 36 Punkte

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`.

Verbindliche Anforderung für die gesamte Hausübung:

Die Verwendung von Streams oder anderen Hilfsklassen ist nicht erlaubt.

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.

H1: Factories ?? Punkte

In Kapitel 06 der FOP 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.

Das generische `public`-Interface `BasicFactory` hat eine Methode `create`, die keine Parameter hat und einen Rückgabetyp vom formalen Typ `T` hat.

Implementieren Sie in allen in Tabelle 1 aufgeführten Klassen (in Package `h09.basic`) das Interface `BasicFactory`. Die Spalte *Klasse* gibt den Namen der Klasse an, die Sie implementieren sollen. Die Spalte `T` gibt den formalen Typ an, mit dem die Implementation `BasicFactory` parametrisieren soll. Die letzte Spalte, **Konstruktor parameter** gibt die Parameter an, die der Konstruktor der Klasse benötigt, um ein Objekt zu erzeugen.

Klasse	T	Konstruktorparameter
`IntegerFactory`	`Integer`	`int start, int step`
`DoubleFactory`	`Double`	`double start, double step`
`StringFactory`	`String`	`int start, String[] text`

Tabelle 1: Übersicht der Klassen, die `BasicFactory` implementieren

H1.1: `IntegerFactory` ?? Punkte

Die überschriebene Methode `create` in `IntegerFactory` soll wie folgt funktionieren: Der erste Aufruf liefert den entsprechenden `start`-Wert der dem Konstruktor übergeben wird. Darauffolgende Aufrufe sollen den zuletzt zurückgegebenen Wert um den `step`-Wert erhöhen und diesen zurückgeben. Nutzen Sie ein Objektattribut namens `current` vom Typ `int` und eine Objektkonstante namens `step` vom Typ `int`, die Sie im Konstruktor initialisieren.

Konkret heißt das, dass der erste Aufruf von `create` den Wert `start` zurückgibt. Der zweite Aufruf von `create` soll den Wert `start + step` zurückgeben. Der dritte Aufruf von `create` soll den Wert `start + 2 * step` zurückgeben, usw.

H1.2: `DoubleFactory` ?? Punkte

Implementieren Sie analog zu `IntegerFactory` die Methode `create` in `DoubleFactory`. Der Unterschied ist, dass diese `create` Methode einen `Double`-Wert zurückgeben soll. Nutzen Sie ein Objektattribut namens `current` vom Typ `double` und eine Objektkonstante namens `step` vom Typ `double`, die Sie im Konstruktor initialisieren.

[1] `https://de.wikipedia.org/wiki/Fabrikmethode`

H1.3: `StringFactory` ?? Punkte

Die `StringFactory`-Klasse funktioniert etwas anders als die anderen beiden. In dieser Implementation entspricht der übergebene `start`-Wert dem Index des zu verwendenden Strings im `text`-Array. Nutzen Sie ein Objektattribut namens `current` vom Typ `int` und eine Objektkonstante namens `text` vom Typ `String[]`, die Sie im Konstruktor initialisieren.

Bei jedem Aufruf soll der nächste String aus dem `text`-Array zurückgegeben werden. (Der erste Aufruf liefert also den String an der Stelle `start` im `text`-Array.) Inkrementieren Sie bei jedem aufruf von `create` den index um `1`. Wenn der Index größer als die Länge des `text`-Arrays ist, soll der Index wieder auf `0` gesetzt werden.

Beispiel:

StringFactory: Beispiel

```
StringFactory factory = new StringFactory(
    1,
    new String[]{"Hallo", "Welt", "!"}
);
System.out.println(factory.create()); // Gibt "Welt" aus
System.out.println(factory.create()); // Gibt "!" aus
System.out.println(factory.create()); // Gibt "Hallo" aus
System.out.println(factory.create()); // Gibt "Welt" aus
System.out.println(factory.create()); // Gibt "!" aus
System.out.println(factory.create()); // Gibt "Hallo" aus
```

H1.4: Traits für die binären Operatorenklassen ?? 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`.

Erweitern Sie also das `public`-Interface `BasicBinaryOperations`, mit den generischen Typparametern `X` und `Y`. Scheiben Sie zwei Methoden namens `add` und `mul`

Die Methode `add` hat zwei Parameter vom formalen Typ `X` und Rückgabetyp `X`. Die Methode `mul` hat einen ersten Parameter vom formalen Typ `X`, einen zweiten Parameter vom formalen Typ `Y` und Rückgabetyp `X`.

Erweitern Sie die Klasse `IntegerBasicBinaryOperations`, die das Interface `BasicBinaryOperations` implementiert, wobei `X` und `Y` die Rolle von `Integer` einnehmen. Die zwei Methoden sind mittels der in Java eingebauten Operatoren für Addition beziehungsweise Multiplikation von Werten primitiver Zahlentypen implementiert.

Erweitern Sie analog zu `IntegerBasicBinaryOperations` die Klasse `DoubleBasicBinaryOperations`, wobei `X` und `Y` die Rolle von `Double` einnehmen. Die zwei Methoden sind wiederum mittels der in Java eingebauten Operatoren für Addition beziehungsweise Multiplikation von Werten primitiver Zahlentypen implementiert.

Erweitern Sie die Klasse `StringBasicBinaryOperations`, die das Interface `BasicBinaryOperations` 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 der FOP). Die Methode `mul` liefert einen

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

leeren String (nicht `null`!) zurück, falls der eingekapselte Wert n des zweiten aktualen Parameters nichtpositiv (also ≤ 0) ist, andernfalls die Konkatenation von n Kopien des ersten aktualen Parameters.

Hier heißt n Kopien des ersten aktualen Parameters, dass der erste Parameter n mal hintereinander wiederholt wird und zurückgegeben wird. Sie dürfen Methoden der Klasse `String` und `Math` verwenden, um diese Aufgabe zu lösen.

Beispiel:

StringBasicBinaryOperations: Beispiel

```java
StringBasicBinaryOperations op = new StringBasicBinaryOperations();
System.out.println(op.add("Hallo", "Welt")); // Gibt "HalloWelt" aus
System.out.println(op.mul("Hallo", 3)); // Gibt "HalloHalloHallo" aus
System.out.println(op.mul("Hallo", 0)); // Gibt "" aus
System.out.println(op.mul("Hallo", -1)); // Gibt "" aus
```

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] Das Interface `StringBasicBinaryOperations` modellieren diesen Fall in ihren beiden generischen Typparametern. Die beiden Klassen `Integer`- und `DoubleBasicBinaryOperations` 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 ?? Punkte

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 Referenz `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) ?? Punkte

In Package `h09.operator.primitive` befinden sich die bereits implementierten Klassen, die Sie in der Hausübung 07 (H2.1, H2.3 und H2.4) implementiert haben:

- `ComposedDoubleBinaryOperator`
- `DoubleMaxOfTwoOperator`
- `DoubleSumWithCoefficientsOperator`

Erweitern Sie in Package `h09.operator` die drei Klassen mit dem gleichen Namen wie in Package `h09.operator.primitive`, aber nun implementieren diese Klassen nicht mehr `DoubleBinaryOperator`, sondern `BinaryOperator`. Instanziieren Sie dabei den Typ `T` von `BinaryOperator` mit `Double`.

> **Hinweis:**
>
> Die Aufrufe von `applyAsDouble` in den Klassen im Package `h09.operator.primitive`, müssen Sie durch Aufrufe von `apply` ersetzen.

> **Hinweis:**
>
> Die Attribute und Konstruktoren müssen auch entsprechend angepasst werden, damit Sie nicht mehr auf `DoubleBinaryOperator` oder primitive `double`-Werte basieren.

H2.2: Verbesserte Version von H2.1 - `ComposedBinaryOperator` ?? Punkte

Sie merken vielleicht, dass die Klasse `ComposedDoubleBinaryOperator` die eben in der H2.1 implementiert wurde, zwar auf `Double` beschränkt ist, aber keine spezielle Funktionalität benutzt, die auf `Double` basiert. Daher können Sie diese Klasse nun generisch implementieren - ohne Einschränkung auf `Double`.

Erweitern Sie die Klasse `ComposedBinaryOperator` in Package `h09.operator` analog zu `ComposedDoubleBinaryOperator` in Package `h09.operator.primitive`, aber nun generisch mit Typparameter `T`.

H2.3: Verbesserte Version von H2.1 - `MaxOfTwoOperator` ?? Punkte

Wie bei `ComposedBinaryOperator` aus der H2.2, ist es auch möglich, die Klasse `DoubleMaxOfTwoOperator` generisch zu implementieren. Überlegen Sie sich, ob es für die Funktionalität der Klasse wirklich nötig ist, sich auf `Double`-Werte zu verlassen.

Erweitern Sie die Klasse `MaxOfTwoOperator` in Package `h09.operator` analog zu `DoubleMaxOfTwoOperator` in Package `h09.operator.primitive`, aber nun generisch mit Typparameter `T`. Nehmen Sie dabei die Einschränkung `Comparable` bei `T` an, wobei `Comparable` selber auf `T` und alle Supertypen von `T` beschränkt ist. Nutzen Sie die Methode `compareTo` aus dem Interface `Comparable`.

H2.4: Verbesserte Version von H2.1 - `SumWithCoefficientsOperator` ?? Punkte

Es ist weiterhin möglich, wie bei der H2.2 und H2.3, die Klasse `DoubleSumWithCoefficientsOperator` so zu gestalten, dass sie generisch ist - diesmal mithilfe der `BasicBinaryOperations` aus der H1.4.

Erweitern Sie in Package `h09.sequence.operator` die Klasse `SumWithCoefficientsOperator`, die `BinaryOperator` implementiert analog zu `DoubleSumWithCoefficientsOperator`, aber nun Generisch mit Typparameter `X` und `Y` ohne ohne `Double` im Namen. Beide Typparameter `X` und `Y` sind uneingeschränkt und der generische Typ `T` von `BinaryOperator` wird mit `X` instanziiert.

Der Konstruktur von `SumWithCoefficientsOperator` verfügt über drei Parameter: `BasicBinaryOperations<X, Y> op`, `Y coeff1`, `Y coeff2`. Zu jedem dieser Parameter gibt es eine entsprechende Objektkonstante, die Sie im Konstruktor mit dem übergebenen Parameter initialisieren müssen.

Nutzen Sie in der `apply`-Methode die Objektkonstanten und -methoden `add` und `mul` aus dem Interface `BasicBinaryOperations`, um die Berechnung der Summe mit den Koeffizienten analog zu `DoubleSumWithCoefficientsOperator` durchzuführen.

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

> **Unbewertete Verständnisfrage:**
>
> Warum sind die Einschränkungen in den drei Klassen unterschiedlich? Was würde es bedeuten wenn das `T` von `MaxOfTwo` stattdessen auf `Number` und alle Subtypen von `Number` eingeschränkt wäre?

H3: Eins nach dem Anderen - Sequences ?? Punkte

Das funktionale Interface `Sequence` (oder Sequenz) stellt eine Folge von Elementen dar. Dieses Interface hat nur eine abstrakte Methode `Iterator<T> iterator()`, die ein `Iterator` zurückgibt, mit dem über die Elemente der Sequenz iteriert werden kann.

H3.1: Von Array zu Sequence ?? Punkte

Um eine `Sequence` zu erstellen, müssen die originalen Elemente gespeichert werden. Dies erfolgt in der Klasse `ArraySequence`.

Erweitern Sie in Package `h09.sequence` die generische Klasse `ArraySequence`, die das Interface `Sequence` implementiert. `ArraySequence` soll parametrisiert sein mit einem generischen Parameter `T` und `Sequence` mit diesen instanziieren. Die Elemente der Sequenz werden in einem Array vom generischen Typ `T` gespeichert, der dem Konstruktor übergeben und direkt einem `values` attribut zugewiesen wird.

Die Methode `iterator()` soll bei jedem Aufruf eine neue Instanz von `Iterator` zurückgeben, der über die Elemente vom Array iteriert. Diese Instanz wird durch eine anonyme Klasse realisiert, die `Iterator` implementiert. Speichern Sie den aktuellen Index in einem privaten `int`-Objektattribut mit dem Namen `index` in der anonymen Klasse. In der anonymen Klasse haben Sie zugriff auf die Objektattribute der `ArraySequence` Klasse.

Die Methode `hasNext()` soll `true` zurückgeben, wenn es in `values` noch weitere Elemente gibt, die mit `next()` abgerufen werden können. Die Methode `next()` soll das nächste Element zurückgeben und den Index um eins erhöhen. Die Methode `next` liefert bei dem ersten Aufruf das Element an Index `0` von `values` zurück. Der zweite Aufruf liefert das Element an Index `1` von `values` zurück usw.

Sobald die Klasse `ArraySequence` fertig implementiert ist, kommentieren Sie im Interface `Sequence` den Rumpf der Methode `of(T...)` ein.

Verbindliche Anforderungen:

- Sie verwenden keine Schleifen und keine Rekursion.

H3.2: 1, 1, 2, 3, 5 - Sequence! ?? Punkte

Die *Fibonacci-Zahlen*[3] sind eine Folge von natürlichen Zahlen, die wie folgt rekursiv definiert ist:

$$fib(0) = 1$$
$$fib(1) = 1$$
$$fib(n) = fib(n-1) + fib(n-2)$$

Erweitern Sie die Klasse `FibonacciSequence`, die `Sequence` implementiert. Instanziieren Sie dabei den generischen Typ `T` von `Sequence` mit `Integer`. Die Klasse hat keinen expliziten Konstruktor, sondern nur den impliziten `public`-Konstruktor ohne Parameter.

Analog zu `ArraySequence` soll die Methode `iterator()` eine neue Instanz von `Iterator` zurückgeben, die über die Elemente der Sequenz iteriert. Allerdings sind diese Elemente nicht wie bei `ArraySequence` vorgespeichert, sondern müssen sie bei jedem Aufruf von `next()` berechnet werden. Der `Iterator` wird analog zu `ArraySequence` durch eine anonyme Klasse realisiert, die `Iterator` implementiert. Legen Sie dafür in der anonymen Klasse zwei Objektattribute vom Typ `int` an: `current` und `next` die mit 0 beziehungsweise 1 initialisiert werden.

Implementieren Sie die `next()` Methode so, dass sie bei jedem Aufruf immer die aktuelle Fibonacci-Zahl zurückgibt und die Attribute `current` und `next` aktualisiert. Der neue `next`-Wert wird immer aus den beiden Attributen per Addition berechnet, ohne das bei jedem `next()`-Aufruf die vorherigen Fibonacci-Zahlen neu berechnet werden. Es gibt keinen Fall, wo der `FibonacciSequenceIterator` keine weiteren Elemente hat.

Der erste Aufruf von `next()` soll also 1 zurückgeben, der zweite Aufruf 1, der dritte Aufruf 2, der vierte Aufruf 3, dann 5, 8, 13, 21, 34, …

Verbindliche Anforderungen:

- Sie verwenden keine Schleifen und keine Rekursion.
- In der von `iterator()` zurückgegebenen `Iterator`-Instanz dürfen Sie keine Methodenaufrufe verwenden.

Anmerkung:

Wenn Sie alle Anforderungen erfüllen, laufen die Methoden `Iterator#hasNext()` und `Iterator#next()` in der Klasse `FibonacciSequence` in konstanter Zeit.

[3] `https://de.wikipedia.org/wiki/Fibonacci-Folge`

H3.3: Basically a Sequence ?? Punkte

In H1.1 - H1.3 haben Sie bereits das Interface `BasicFactory` implementiert. Nun würden wir gerne anhand einer `BasicFactory` eine `Sequence` erstellen. Erweitern Sie dafür in Package `h09.sequence` die generische Klasse `BasicFactorySequence`, die `Sequence` implementiert. `BasicFactorySequence` soll einen generischen Typen von `T` haben und `Sequence` mit diesen instanziieren. Die Klasse soll einen Konstruktor haben, der eine Instanz von `BasicFactory` als Parameter entgegennimmt, wobei der generische Typ `T` von `BasicFactory` mit dem generischen Typ `T` von `Sequence` übereinstimmt, und direkt als Objektkonstante namens `factory` mit dem gleichen Typ speichert.

Analog zu H3.1 und H3.2, soll die Methode `iterator()` eine neue Instanz von `Iterator` zurückgeben, die über die Elemente der Sequenz iteriert und durch eine anonyme Klasse realisiert wird. Diese Elemente sind wie bei der H3.2 nicht vorgespeichert, sondern müssen bei jedem Aufruf von `next()` durch die übergebene `BasicFactory` Instanz erzeugt werden.

Implementieren Sie die `next()`-Methode so, dass sie bei jedem Aufruf die `create()`-Methode der `BasicFactory`-Instanz aus `BasicFactorySequence` aufruft und das zurückgegebene Objekt zurückgibt. Es gibt keinen Fall, wo der `BasicFactorySequenceIterator` keine weiteren Elemente hat.

Sobald die Klasse `BasicFactorySequence` fertig implementiert ist, kommentieren Sie im Interface `Sequence` den Rumpf der Methode `of(BasicFactory<T>)` ein.

H4: Sequence operationen ?? Punkte

Nun würden wir gerne einige Operationen auf `Sequence` durchführen. Diese Operationen sind in Klassen realisiert, die `Sequence` implementiert. Es sind bereits zwei solche Operationen vorgegeben:

- Die Klasse `LimitSequence` bekommt eine `Sequence` und eine maximale Anzahl an Elementen übergeben, und liefert nur die ersten `n` Elemente der `Sequence` zurück.
- Die Klasse `OnEachSequence` bekommt eine `Sequence` und einen `Consumer` übergeben, und ruft für jedes Element der `Sequence` die `accept` Methode von dem `Consumer` auf.

Beispiel:

LimitSequence

```java
Sequence<String> originalSeq = Sequence.of("a", "b", "c", "d", "e");
Sequence<String> limitedSeq = new LimitSequence<>(originalSeq, 2);
Sequence<String> onEachSeq = new OnEachSequence<>(
    limitedSeq, s -> System.out.println("OnEach: " + s));
Iterator<String> it = onEachSeq.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

Output

```
$ OnEach: a
$ a
$ OnEach: b
$ b
```

H4.1: FilteringSequence ?? Punkte

Implementieren Sie die Klasse `FilteringSequence` in Package `h09.sequence`. Die Klasse hat einen generischen Typparameter `T`, welcher den Typ der Elemente der `Sequence` angibt und soll `Sequence` implementieren und dabei den generischen Typ `T` verwenden. Dafür müssen Sie zunächst zwei private Objektkonstanten definieren, die in der gleichen Reihenfolge dem Konstruktor übergeben werden und direkt in den Objektkonstanten gespeichert werden:

- Das Attribut `sequence` vom Typ `Sequence`, welches die zu filternde Sequenz speichert.
- Das Attribut `predicate` vom Typ `Predicate`, welches für jedes Element entscheidet, ob es in der Sequenz enthalten sein soll.

Ergänzen Sie die Typen beider Attribute, sodass für feststehendes `T` die maximal mögliche Kompatibilität zwischen den Typen der Attribute und des Interfaces erreicht wird. Das heißt besonders bei `predicate`, dass `T` und alle superklassen von `T` gelten.

Die `iterator()`-Methode funktioniert ähnlich wie in der H3, allerdings wird hier die „Eingabe“ durch die `Sequence` realisiert, die im Konstruktor übergeben wurde. Als Erstes muss in der anonymen Klasse in der `iterator()`-Methode eine Objektkonstante namens `iterator` vom Typ `Iterator` definiert werden, die direkt von einem Aufruf von `sequence.iterator()` initialisiert wird. Achten Sie dabei darauf, dass der Typparameter von `Iterator`, wie auch der Typparameter des Attributes von `sequence` in `FilteringSequence`, auf `T` und Unterklassen von `T` eingeschränkt ist.

Die `Iterator`-Implementation in der `iterator()`-Methode funktioniert so, dass er bei jedem Aufruf der Methode `hasNext()`, falls das Attribut `next` nicht auf ein Objekt zeigt, den gespeicherten `iterator` durchsucht, bis er ein Element findet, welches vom `predicate` akzeptiert wird. Wird ein Element gefunden, wird dieses in dem Objektattribut `next` gespeichert und die Methode `hasNext()` liefert `true` zurück. Sollte es kein solches Element geben, so gibt die Methode `hasNext()` `false` zurück.

Die Methode `next()` liefert das gespeicherte Element zurück und setzt `next` auf `null`. Sie können davon ausgehen, dass vor jedem Aufruf von `next()` die Methode `hasNext()` aufgerufen wurde und diese `true` zurückgegeben hat.

Beispiel:

FilteringSequence: Gerade Fibonacci-Zahlen

```
Sequence<Integer> seq = new FilteringSequence<>(
    new FibonacciSequence(),
    x -> x % 2 == 0
);
Sequence<Integer> limitedSeq = new LimitSequence<>(seq, 6);
Iterator<Integer> it = limitedSeq.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

Output

```
$ 2
$ 8
$ 34
$ 144
$ 610
$ 2584
```

H4.2: Von T nach R - Transformation ?? Punkte

Implementieren Sie die Klasse `TransformingSequence` in Package `h09.sequence`. Die Klasse hat zwei generische Typparameter `T` und `R` wobei `T` den Typ der Elemente der „Eingabe“-`Sequence` angibt und `R` den Typ der Elemente der „Ausgabe“-`Sequence` angibt. In dieser Klasse geht es darum Elemente vom Typ `T` in Elemente vom Typ `R` zu transformieren.

Die Klasse `TransformingSequence` stellt wie `FilteringSequence` eine `Sequence` dar – und implementiert dementsprechend das Interface `Sequence` – allerdings wird hier der generische Parameter von `Sequence` nicht durch `T` sondern durch `R` instanziiert. Grund dafür ist, dass die `TransformingSequence` schließlich einen `Iterator` erzeugen soll, der transformierte Elemente vom Typ `R` zurückgibt.

Als Erstes müssen zwei private Objektkonstanten definiert werden, die in der gleichen Reihenfolge dem Konstruktor übergeben werden und direkt in den Objektkonstanten gespeichert werden:

- Das Attribut `sequence` vom Typ `Sequence`, welches die zu transformierende Sequenz speichert.
- Das Attribut `function` vom Typ `Function`, welches für jedes Element die Transformation angibt.

Ergänzen Sie die Typen beider Attribute, sodass für feststehendes `T` und `R` die maximal mögliche Kompatibilität zwischen den Typen der Attribute und des Interfaces erreicht wird. Das heißt besonders bei `function`:

- Bei `T`, dass `T` und alle Supertypen von `T` gelten.
- Bei `R`, dass `R` und alle Subtypen von `R` gelten.

Speichern Sie wie in der H4.1 die `Iterator` Instanz in einer Objektkonstante in der anonymen Klasse der `iterator()`-Methode die direkt von einem Aufruf von `sequence.iterator()` initialisiert wird. Die `hasNext()`-Methode liefert genau dann `true` zurück, wenn der gespeicherte `Iterator` noch ein Element hat. Die `next()`-Methode liefert das Ergebnis des Aufrufs von `function.apply()` auf dem nächsten Element des gespeicherten `Iterator` zurück.

Beispiel:

TransformingSequence: Quadrat der Fibonacci-Zahlen

```
Sequence<String> seq = new TransformingSequence<>(
    new FibonacciSequence(),
    x -> "Next number: " + x * x
);
Sequence<Integer> limitedSeq = new LimitSequence<>(seq, 6);
Iterator<String> it = limitedSeq.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

Output

```
$ Next number: 1
$ Next number: 1
$ Next number: 4
$ Next number: 9
$ Next number: 25
$ Next number: 64
```

H4.3: FlatteningTransformingSequence ?? Punkte

Implementieren Sie die Klasse `FlatteningTransformingSequence` in Package `h09.sequence`. Die Klasse ist so aufgebaut, wie die Klasse `TransformingSequence` mit dem folgenden Unterschied:

- Das Attribut `function` hat als Ergebnistypen statt `R` eine `Sequence` von `R`

Achten Sie bei der Festlegung des Typen von `function` erneut darauf einen maximal möglichen Grad an Freiheit zu erlauben.

Die Funktionlität der Klasse `FlatteningTransformingSequence` unterscheidet sich zur Klasse `TransformingSequence` darin, dass durch die Transformation eine „Flattening“ Operation ausgeführt wird. Eine `Flattening` Operation bildet dabei eine zweifach genestete Datenstruktur auf eine nicht genestete Datenstruktur ab. Es wird also z.B. eine Liste von Listen auf eine einfache Liste abgebildet.

Speichern Sie wie zuvor in der anonymen Klasse in der `iterator()`-Methode die `Iterator` Instanz in einer Objektkonstante, die direkt von einem Aufruf von `sequence.iterator()` initialisiert wird. Erstellen Sie ein weiteres privates Objektattribut namens `currentIterator` vom Typ `Iterator` wobei der generische Typ von `Iterator` durch `R` und alle Subtypen von `R` Instanziiert wird. Initialisieren Sie `currentIterator` mit `null`.

Implementieren Sie die `hasNext()`-Methode durch eine `while`-Schleife, die so lange `currentIterator` keine Elemente hat, diese mithilfe von `function` neu initialisiert und `currentIterator` mit dem neuen `Iterator` überschreibt. Hat der `currentIterator` Elemente, so liefert die `hasNext()`-Methode `true` zurück. Sollte der `currentIterator` keine Elemente mehr haben, und es gibt keine weiteren Elemente in dem gespeicherten `iterator` der anonymen Klasse, so liefert die `hasNext()`-Methode `false` zurück.

Die `next()`-Methode liefert das nächste Element des `currentIterator` zurück.

Sie können davon ausgehen, dass vor jedem Aufruf von `next()` die Methode `hasNext()` aufgerufen wurde und diese `true` zurückgegeben hat.

Beispiel:

FlatteningTransformingSequence: Doppelte Werte einer Liste von Listen

```
Sequence<String> ogSeq = Sequence.of("1", "23", "456");
Sequence<Character> charSeq = new FlatteningTransformingSequence<>(
    ogSeq, s -> PrimitiveSequence.of(s.toCharArray()));
Sequence<Integer> seq = new TransformingSequence<>(
    charSeq, Character::getNumericValue);
Sequence<Integer> limitedSeq = new LimitSequence<>(seq, 6);
Iterator<Integer> it = limitedSeq.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

Output

```
$ 1
$ 2
$ 3
$ 4
$ 5
$ 6
```

H4.4: Einfachere Syntax bei Verwendung von Sequences ?? Punkte

Bis jetzt wurde zum Erzeugen einer Sequenz immer der Konstruktor der entsprechenden Klasse explizit aufgerufen. Das ist zwar nicht unbedingt schlecht, aber wird bei mehreren Verkettungen schnell unübersichtlich. Wir nehmen als Beispiel den folgenden Code:

Verkettung von Sequenzen mit Konstruktoren

```
Sequence<String> seq0 =
    Sequence.of("Generics", "sind", "nicht", "toll", "und", "super");

Sequence<String> seq1 = new FilteringSequence<>(seq0, s -> !s.equals("nicht"));
Sequence<String> seq2 = new TransformingSequence<>(seq1, String::toUpperCase);
Sequence<String> seq3 = new TransformingSequence<>(seq2, s -> s + "!");
Sequence<String> seq4 = new LimitSequence<>(seq3, 3);
```

In diesem Beispiel wird zuerst eine `Sequence` vom Typ `String` erstellt. Diese wird dann durch mehrere Verkettungen von Sequenzen transformiert. Die Verkettung erfolgt dabei immer durch den Aufruf des Konstruktors der jeweiligen Sequenz, wobei das Ergebnis aus der letzten Operation als Argument übergeben wird. Diese `Sequence` hat am Ende den folgenden Inhalt:

Output

```
$ GENERICS!
$ SIND!
$ TOLL!
```

Es ist aber möglich diese Verkettung ohne zusätzliche Variablen zu schreiben – mithilfe der `then`-Methode der `Sequence`-Klasse. Diese Methode nimmt eine Funktion als Argument, die eine Sequenz vom Typ `T` erwartet und liefert eine Sequenz vom Typ `R` zurück. Damit ist es möglich, die Verkettung von Sequenzen wie folgt zu schreiben:

Verkettung von Sequenzen mit `then()` und `of()`

```
Sequence<String> seq =
    Sequence.of("Generics", "sind", "nicht", "toll", "und", "super")
        .then(FilteringSequence.of(s -> !s.equals("nicht")))
        .then(TransformingSequence.of(String::toUpperCase))
        .then(TransformingSequence.of(s -> s + "!"))
        .then(LimitSequence.of(3));
```

Die Semantik von beiden Varianten ist identisch, allerdings ist die zweite wesentlich kürzer, übersichtlicher und weniger fehleranfällig, weil die Referenzen auf die Sequenzen nicht explizit verwaltet werden müssen.

Die Methode `then` in Interface `Sequence` ist bereits implementiert. Zusätzlich sind die Klassenmethode `of` in `LimitSequence` und `OnEachSequence` bereits als Beispiel implementiert.

Implementieren Sie die Klassenmethoden `of` in `FilteringSequence`, `TransformingSequence` und `FlatteningTransformingSequence`. Die Parameter der Methoden sind dieselben wie beim Konstruktor der entsprechenden `Sequence`Implementation, allerdings ohne die Sequenz selbst. Achten Sie darauf, dass die Methoden `of` in den Klassen `TransformingSequence` und `FlatteningTransformingSequence` zwei generische Typparameter haben.

H5: Collect·e – Der Letzte räumt die Sequenz auf ?? Punkte

Um die Elemente einer modifizierten Sequenz letztendlich ausgeben zu können, implementieren Sie drei Klassen, die das generische Interface `SequenceCollector<T,R>` implementieren.

`SequenceCollector<T,R>` hat eine Objektmethode `collect`, die eine Sequenz von `T` gegeben bekommt und die Elemente dieser zu einem Objekt von `R` zusammenfasst und zurückliefert.

H5.1: Eine listige Idee ... ?? Punkte

Erweitern Sie die generische Klasse `ToListCollector`, die das Interface `SequenceCollector` implementiert. Die Klasse `ToListCollector` hat einen Typparameter `T` und instanziiert `SequenceCollector` mit diesen. Denken Sie daran, dass der generische Typ `R` von `SequenceCollector` mit dem Ergebnis der Methode `collect` übereinstimmen muss. Die `collect`-Methode sammelt die Elemente der gegebenen Sequenz in eine neue Liste von `T` und liefert diese zurück. Die Reihenfolge der Elemente in der Liste soll gleich der Reihenfolge sein, in der die Elemente vom Iterator der Sequenz ausgegeben werden. Das Erste vom Iterator gelieferte Element soll also das erste Element dieser Liste sein.

> **Hinweis:**
>
> Benutzen Sie als Implementation von `java.util.List` die Klasse `ArrayList`, die Ihnen aus Kapitel 07 der FOP bekannt ist.

H5.2: Sum, Sum, Sum, ... ne Sequence fliegt herum ?? Punkte

Deklarieren Sie Klasse `SummingCollector` so um, dass diese einen unbeschränkten Typparameter `T` besitzt und das Interface `SequenceCollector` so implementiert, dass Methode `collect` eine Sequenz von `T` oder einem Subtyp von `T` gegeben bekommt und ein Element von `T` liefert.

Erstellen Sie zuerst einen Konstruktor, der als ersten Parameter den *Initialwert* und als zweiten Parameter die *Verknpüfungsoperation* gegeben bekommt. Der Initialwert soll von Typ `T` sein. Bei der Verknpüfungsoperation handelt es sich um ein Objekt von `BasicBinaryOperations`, welches auf Objekte von Typ `T` angewendet werden kann und ein Objekt von Typ `T` liefert.

Implementieren Sie Methode `collect` so, dass die Elemente der gegebenen Sequenz „summiert“ werden. Hierzu wenden Sie das *Prinzip der Faltung* an, welches Sie unter anderem aus Kapitel 04c, ab Folie 139 der FOP kennen. `collect` nutzt als ersten Zwischenwert den dem Konstruktor gegebenen Initialwert. Der Zwischenwert wird solange durch die Summe des aktuellen Zwischenwerts und des nächsten Werts des Iterators ersetzt, bis der Iterator keine Elemente mehr liefert. Der letzte Zwischenwert ist der Wert, welcher von `collect` zurückgeliefert wird.

> **Hinweis:**
>
> Sie können sich bei Ihrer Implementation an Methode `fold` aus Kapitel 07, ab Folie 100 der FOP orientieren.

H5.3: Collecteur Binaire ?? Punkte

Deklarieren Sie Klasse `BinaryFoldCollector` genauso wie in H5.2 um. Die `collect`-Methode in Klasse `BinaryFoldCollector` arbeitet funktional identisch zu `SummingCollector`, arbeitet aber mit mit `BinaryOperator` anstelle von `BasicBinaryOperator`.

Implementieren Sie in `BinaryFoldCollector` wie in H5.2 einen Konstruktor und `collect` mit dem im letzten Absatz erwähnten Unterschied.

> **Unbewertete Verständnisfrage:**
>
> Wann kann die Verwendung eines *spezielleren* Interface wie `BinaryArithmeticOperator` sinnvoll sein?