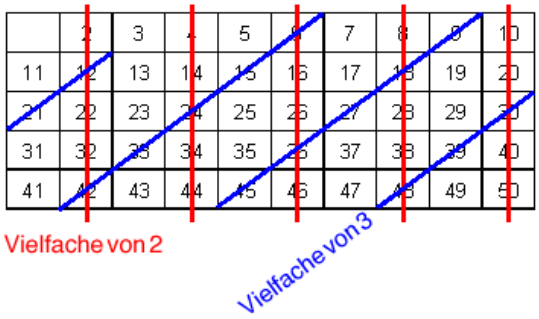
Programmieren 2 (INF)
Übungsblatt 2

In diesem Übungsblatt geht es um verschiedene Konzepte der Typisierung von Objekten in Java: *Klassen, abstrakte Klassen und Interfaces*. Am Beispiel von Zahlenfolgen wird eine komplexe Klassenstruktur entwickelt.



Aufgabe 1 (Projekt einrichten)

Richten Sie zunächst (falls noch nicht geschehen) in Ihrer IDE ein Java-Projekt ein, welches die Quelldateien für Übungen, Demos etc. zu dieser Vorlesung aufnehmen soll. Das Wurzelpaket sollte Ihren Namen tragen (bzw. Ihre Initialen), darin sollten verschachtelt die Pakete `prog2` und `exercises` angeordnet werden:

- `<IhrName>.prog2.exercises`

Im Paket `exercises` werden für jedes Übungsblatt neue Pakete eingefügt, z.B. `set01` für das erste Übungsblatt und `set02` für dieses Übungsblatt.

Legen Sie zusätzlich die folgenden Pakete an:

- `rl.prog2.exercises`

In das zugehörige Subverzeichnis `exercises` können Sie dann die Musterlösungen einspielen.

Integrieren Sie auch ihre Lösung vom ersten Übungsblatt in die neue Struktur. Die Funktionen Rename und Move im Refactor-Menü Ihrer IDE helfen Ihnen dabei!

Aufgabe 2

Eine Folge ist eine endliche oder auch unendliche Auflistung von Objekten. Wir beschränken uns hier auf Folgen ganzer Zahlen. Solche Zahlenfolgen sollen nun als Objekte repräsentiert werden.

Legen Sie im Paket `set02` ein Paket `sequences` an und definieren Sie darin den Typ `Sequence`. Er soll lediglich über zwei Operationen verfügen:

- `hasNext()` liefert die Information, ob es ein weiteres Folgenelement gibt.
- `nextElement()` liefert das nächste Element (eine ganze Zahl).

Man erhält also die Objekte einer Folge durch mehrfaches Aufrufen von `nextElement`. Bei endlichen Folgen kann mit der Sicherheitsabfrage `hasNext` verhindert werden, dass man über das Folgenende hinaus Zugriffe versucht. Der Typ soll so allgemein wie möglich definiert werden und für die Implementierung den größtmöglichen Spielraum bieten.

Aufgabe 3

Die natürlichen Zahlen (1, 2, 3, …) sind ein Beispiel für eine solche Folge. Implementieren Sie diese, indem Sie eine Klasse `Naturals` erstellen und dafür sorgen, dass Instanzen der Klasse auch den Typ `Sequence` haben.

Erstellen Sie zum Testen eine Klasse `SequenceTest`. Sie sollte eine statische Methode `println` enthalten, die eine Folge als Parameter akzeptiert und maximal die ersten 10 Elemente der Folge ausgibt. Wenn Sie nun in die Klasse die folgende Methode einfügen,

```
public static void main(String[] args) {
      println(new Naturals());
}
```

könnte die Ausgabe wie folgt aussehen:
`1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...`

Erstellen Sie des Weiteren eine Klasse `Range` zur Repräsentation der natürlichen Zahlen in einem bestimmten Intervall.

`println(new Range(7, 14));`
könnte z.B. zu folgender Ausgabe führen:
`7, 8, 9, 10, 11, 12, 13, 14.`

Aufgabe 4

Ein Filter ist eine spezielle Zahlenfolge, die Zahlen einer gegebenen Zahlenfolge elementweise prüft und nur diejenigen Elemente passieren lässt, die eine bestimmte Bedingung erfüllen.

Filter können alle nach dem gleichen Bauprinzip implementiert werden. Nur die Prüfung der Bedingung muss für jeden Filter separat implementiert werden. Wir realisieren den Test über eine Methode:

- `boolean testPassCondition(int element)`

Geben Sie eine Klassendefinition an, für die selbst keine Instanzen gebildet werden können, die aber alles, was bei der Implementierung von Filtern gemeinsam nutzbar sein könnte, enthält, z.B. die Zahlenquelle - ein Attribut vom Typ `Sequence`.

Tipps zur Realisierung der Klasse `Filter` finden Sie bei Bedarf weiter hinten!

Aufgabe 5

Implementieren Sie einen Filter `Evens`, der nur gerade Zahlen passieren lässt.

`println(new Evens(new Range(7, 14)));`
könnte z.B. zu folgender Ausgabe führen:
`8, 10, 12, 14.`

Realisieren Sie einen weiteren Filter `ZapMultiples`. Er erhält im Konstruktor neben der Zahlenquelle noch eine Basiszahl und lässt nur diejenigen Zahlen passieren, die keine Vielfachen der Basiszahl sind.

`println(new ZapMultiples(new Range(7, 14), 3));`
könnte z.B. zu folgender Ausgabe führen:
`7, 8, 10, 11, 13, 14.`

Warum ist der folgende Aufruf problematisch?

```
println(new ZapMultiples(new Evens(new Naturals()), 2));
```

Aufgabe 6 (Knobelaufgabe – freiwillig)

Das Sieb des Eratosthenes ist ein Algorithmus zur Berechnung von Primzahlen:

(a) Nimm die natürlichen Zahlen und entferne die 1.
(b) Vom Rest entferne die erste Zahl *p*. Diese ist eine Primzahl.
(c) Entferne alle ganzzahligen Vielfachen von *p*.
(d) Fahre fort mit Schritt (b).

Implementieren Sie den Algorithmus mit den zuvor definierten Klassen. Die Implementierung kann sehr elegant als Folge `PrimeNumbers` realisiert werden. Wenige Zeilen Code genügen!

```
println(new PrimeNumbers());
```

könnte z.B. zu folgender Ausgabe führen:

```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ...
```

Aufgabe 7 (Abgabe freiwillig, Besprechung im Tutorium)

Zeichnen Sie ein Klassendiagramm für die gerade erstellte Klassenstruktur.

Tipps

Die größte Schwierigkeit bei der Realisierung der Filter besteht darin, dass die Berechnung des nächsten Folgenwertes nicht erst geschehen kann, wenn die Methode `nextElement()` aufgerufen wird. Die Methode `hasNext()` muss bereits vorher wissen, ob es ein weiteres Element überhaupt gibt. Eine mögliche Lösung besteht in der Ausstattung der Klasse `Filter` mit zusätzlichen Attributen:

- `next` (enthält die Zahl, die `nextElement()` im nächsten Aufruf zurückgibt) und
- `hasNext` (gibt an, ob der Wert von `next` gültig ist oder das Ende der Folge bereits erreicht wurde)

Die Werte dieser Attribute können in einer Methode `computeNext()` aktualisiert werden. Sie sollte sowohl im Konstruktor der abgeleiteten Klassen (in `Filter` selbst wäre schöner, aber es gibt ein Problem…) als auch in der `nextElement()`-Methode der Klasse `Filter` aufgerufen werden. Alles klar?