Aufgabe 6.1 (Geometrische Figuren)

Im Skript wurde eine Klassenhierarchie vorgestellt, welche geometrische Objekte modellieren soll. Die Klasse `Figur` soll nun als *abstrakte* Klasse implementiert werden. Für sie gelten folgende Spezifikationen:

- Sie enthält zwei abstrakte Methoden:

```
void draw()     // Figur zeichnen. Da wir nicht in einer graphischen Oberfläche
                // arbeiten, soll hier nur der Typ des Objekts ausgegeben werden.
double area()   // Ausgabe der Fläche der Figur
```

- Sie enthält folgende konkreten Elemente:
  - Jede Figur besitzt einen Ursprungspunkt in Form zweier Koordinaten im `double`-Format.
  - Für die beiden Koordinaten sollen Getter- und Setter-Methoden bereitgestellt werden.
  - Drei Standard-Konstruktoren (Default, Copy, User-Defined)

Implementieren Sie die drei konkreten Klassen `Kreis`, `Rechteck` und `Linie`, indem Sie von `Figur` ableiten.

- Beschreiben Sie wesentliche Unterschiede zwischen den beiden abstrakten Klassen `Payment` und `Figur`.
- Würde es einen Sinn ergeben, `Figur` durch ein Interface zu ersetzen? Begründen Sie!

Aufgabe 6.2 (`Payment`)

Ersetzen Sie in der Klassenhierarchie unter `Payment` die Basisklasse durch eine abstrakte Klasse und ersetzen Sie geeignete Methoden durch abstrakte Methoden.

- Würde es einen Sinn ergeben, `Payment` durch ein Interface zu ersetzen? Begründen Sie!

Aufgabe 6.3 (Die Klasse `MyString0` nachbearbeiten)

Unsere Klasse `MyString0` (OOP1, Übung 10) ist im Gegensatz zu den Bibliotheks-Klassen `String` und `StringBuilder` nicht standardisiert. Dennoch wurden für die damalige Übung die zu schreibenden Methoden so ausgewählt, dass es nur noch wenig Aufwand erfordert, `MyString0` an gewisse Standards der Bibliothek anzupassen.

a) Sie wissen, dass mit Hilfe von Interfaces in der API einer Klasse Standards durchgesetzt werden können. Finden Sie heraus, welche Interfaces Kandidaten für `MyString0` sein könnten, indem Sie die Dokumentationen der Klassen `String` und `StringBuilder` genau untersuchen.

b) Sie können nun versuchen, `MyString0` zu einer Klasse zu erheben, welche den Standards eines Strings entspricht, indem Sie behaupten, sie implementiere gewisse Interfaces (d.h. Sie ergänzen Ihre Klasse um eine entsprechende `implements`-Klausel).

c) Wie Sie schnell merken werden, ist der Compiler mit Ihrer Aktion nicht einverstanden und meldet Fehler. Die Methode `subSequence()` und die `append()` Methoden erzeugen Fehlermeldungen. Warum?

d) Passen Sie die Signaturen dieser Methoden entsprechend an. Weshalb bekommen Sie immer noch Fehler? Wie können Sie die `append()`-Methoden verändern, so dass die Fehler verschwinden?

e) Es fehlt nun noch das Interface `Comparable`, um die Sammlung zu komplettieren. Es unterscheidet sich dadurch, dass es typisiert werden kann. Deshalb müssen Sie schreiben:

```
implements Comparable<MyString0>
```

Die Implementierung der Methode sieht wieder «normal» aus: `public int compareTo(MyString0 m)...`
Implementieren Sie auch noch den Vergleich zwischen zwei `MyString0`-Objekten.

Aufgabe 6.4 (Ein Interface für Stack-artige Datenstrukturen)

In verschiedenen bisherigen Aufgaben haben Sie die Datenstruktur `Stack` hergestellt, bzw. damit gearbeitet. Die wichtigsten Methoden der Objekte dieser Klasse waren: `push()`, `pushAll()`, `pop()`, `top()`, `isEmpty()` sowie `isFull()`. Es gibt ausser dem Stack noch weitere Datenstrukturen, für welche man dieses Set von Methoden sinnvollerweise voraussetzen könnte (z.B. *Queue* und *Priority Queue*).

Falls man alle diese Datenstrukturen anbieten wollte, wäre es nützlich, für alle ein gemeinsames Interface zu definieren, das dieses Set von Methoden deklariert. Wir gehen auch in dieser Aufgabe davon aus, dass ausschliesslich Objekte des Typs `String` gespeichert werden sollen.

a) Erstellen Sie ein neues Projekt und deklarieren Sie dort das Interface `PushPopCollection`, welches die vorhin aufgezählten Methoden enthalten soll.

b) Kopieren Sie die `Stack`-Klasse aus Übung 4, Aufgabe 4.1 in Ihr neues Projekt. Sorgen Sie dafür, dass diese Stack-Klasse das Interface `PushPopCollection` erfüllt (bzw. implementiert). Sie brauchen dafür nicht sehr viel zu tun, denn alle geforderten Methoden existieren ja bereits. Im Prinzip müssen Sie nur den Klassenkopf von Stack anpassen. Sie können aber noch etwas für das Code-Styling tun, indem Sie die Methoden des Interfaces in der Klasse `Stack` mit `@Override`-Anmerkungen auszeichnen.

Aufgabe 6.5 (Warteschlangen sind auch `PushPopCollections`)

Da Sie nun schon einige Erfahrung mit der Stack-Klasse gesammelt haben, sollte es Ihnen keine grösseren Probleme bereiten, zusätzlich noch die Klasse `Queue` nach dem Muster der Klasse `Stack` zu implementieren. Sie ist im Prinzip gleich aufgebaut wie `Stack`, weist aber einen wesentlichen Unterschied auf: Während beim Stack immer das *letzte* Element, das mit `push()` eingestellt wurde, *zuerst* mit `pop()` wieder zum Vorschein kommt (sogenannte *last-in-first-out* Struktur), wird bei `Queue` mit `pop()` das *älteste* eingestellte Element hervorgeholt (sogenannte *first-in-first-out* Struktur). Deshalb bezeichnet man Queues auch als *Warteschlangen*.

Es gibt verschiedene Wege, dies zu erreichen. Wir verwenden hier eine einfach zu verstehende Strategie, die allerdings nicht zu den schnellsten gehört:

- `push()` arbeitet so, wie Sie es von der `Stack`-Klasse gewohnt sind.
- `pop()` gibt das erste Element, d.h. `elements[0]` zurück. Danach müssen allerdings die verbleibenden Elemente um eine Position verschoben werden, so dass `elements[1]` nun zu `elements[0]` wird, usw. Vergessen Sie nicht, danach `nextFree` um 1 zu verringern!

Schreiben Sie die Klasse `Queue` (am einfachsten ist es, wenn Sie den Inhalt `Stack`-Klasse kopieren und die erforderlichen Anpassungen vornehmen.

Da auch Queue alle Methoden von `PushPopCollection` implementiert, soll sie so eingerichtet werden, dass dies auch der Compiler weiss (*siehe Aufgabe 6.4*).

Aufgabe 6.6 (Testen der beiden Klassen, Polymorphie)

a) Testen Sie zunächst, ob `Queue` korrekt arbeitet.

b) Sie sollen nun einen Polymorphie-Test mit den beiden Klassen durchführen. Legen Sie hierzu einen Array `test` vom Typ `PushPopCollection` mit 2 Elementen an. Dem ersten Element von `test` weisen Sie ein `Stack`-Objekt, dem zweiten Element ein `Queue`-Objekt zu.

   Füllen Sie beide Datenstrukturen mit denselben Elementen in derselben Reihenfolge auf und lassen Sie sich den Inhalt zur Kontrolle anzeigen. Anschliessend entfernen Sie mit `pop()` je ein Element aus den beiden Datenstrukturen. Wenn Sie sich jetzt die beiden Strukturen anzeigen lassen, sollten die Inhalte unterschiedlich sein!

Sie haben nun `pop()` jedesmal auf einer `PushPopCollection`-Referenz aufgerufen, und jedesmal wurde die richtige Version von `pop()` ausgewählt. Wie Sie sehen, funktioniert die Polymorphie auch perfekt mit Referenzen auf ein Interface.