6 Verkettete Liste im Einsatz und Typenparameter

6.1 Highscore Revisited

Nun wollen wir eine **verkettete Liste** einsetzen. Und zwar mal wieder für Ihre `Highscore`-Implementation.

Fühlen Sie Sich frei Ihre eigene Implementation von `LinkedList` zu verwenden *(Übung 5.2)*. Ansonsten steht die vordefinierte `LinkedList` von Java zur Verfügung *(wie wir Sie in Serie 2, `MixMiddle` verwendet haben)*. Wenn Sie Ihre eigene `LinkedList` verwenden möchten, sind Sie entweder auf `int` beschränkt, oder Sie bauen Ihren Code zuerst zu einer generischen Implementation um *(freiwillig, siehe Übung 6.3 unten)*. Wenn `LinkedList` generisch ist, können wir sie ohne Änderungen z. B. mit den Klassen `Player` *(Übung 3.4)*, wie auch `Integer` verwenden.

Da Sie statt dem Array nun eine verkettete Liste verwenden, **entfällt** das Verschieben der nachfolgenden Scores beim Einfügen einer neuen Punktezahl. Passen Sie Ihre `Highscore`-Klasse entsprechend an und testen Sie das Resultat.

6.2 Komplexität

Sie müssen zu dieser Aufgabe **keinen** Code schreiben. Es geht nur um eine **theoretische** Betrachtung. Gehen Sie davon aus, dass das Sortieren mit **Bubble Sort** geschieht.

Das **Einfügen** eines neuen Eintrags in den `Highscore` lässt sich weiter "vereinfachen": **Falls** die neue Punktzahl ausreicht um einen Highscore-Platz zu erreichen, überschreiben wir prinzipiell den letzten (untersten) Eintrag und lassen die Liste anschliessend sortieren.

Halten Sie diese Lösung für korrekt? Begründen Sie Ihre Antwort.

Berechnen und vergleichen Sie die **Komplexität** der beiden Lösungen. Verwenden Sie die **Gross-O-Notation**.

6.3 Generische Verkettete Liste

Freiwillig: **Typenparameter** (auch "**Generics**") sind eine praktische Einrichtung von Programmiersprachen, haben aber nichts mit Objektorientierung (OOP) zu tun. Typenparameter und OOP existieren orthogonal zueinander. Sie lassen sich gut kombinieren.

Ziel ist es eine Klasse wie die verkettete Liste **universell** (**generisch**) zu gestalten. Das heisst, verkettete Listen müssen bezüglich Nutzdaten nicht auf `int`-Zahlen beschränkt sein. Dank **Typenparametern** können wir eine **beliebige Klasse** als **Typ** für die **Nutzdaten** einsetzen.

Typenparameter werden in **spitzen Klammern** "<T>" geschrieben. "`T`" steht als **"Platzhalter"** für den Namen der Klasse.

Listing 1: ListElement.java

```
1 public class ListElement<T>
2 {   public T data;
3     public ListElement<T> next; // reference of the next element in the list
4     ...
```

Vorgehen:

a) In Eclipse sehen Sie auf der linken Seite Ihre bereits vorhandenen Projekte. Praktischerweise können Sie dort mittels **Copy & Paste** (Ctrl.+C & Ctrl.+V) Kopien ganzer Projekte anlegen. Kopieren Sie Ihr Projekt `LinkedList` in ein neues Projekt z. B. "`GenericLinkedList`".

b) Die Klasse `ListElement` ist schnell angepasst. Wenn Sie den Typenparameter wie in *Listing 1 auf Zeile 1* eingeführt haben, können Sie in der ganzen Klasse einfach sämtliche Vorkommen von `int` durch `T` ersetzen *(wie in Zeile 2)* und die Vorkommen von `ListElement` mit "<T>" ergänzen. Dies gilt jedoch nicht für die Konstruktor-Signatur (Kopfzeile) "`public ListElement(T dat ...`" *(ohne <T>)*!

c) Genau gleich sind die Änderungen bei `LinkedList<T>`. Passen Sie aber auf, dass Sie `nicht` auch die Index-Variablen "pos" als vom Typ `T` definieren. Der Index (Element-Zähler) bleibt vom Typ `int`!

d) Der Konstruktor von `LinkedList` initialisiert seine Felder vom Typ `ListElement` neu mit "`null`" anstelle von 0. Weil 0 ist der Default-Wert von `int`-Variablen. Da `T` aber für eine Klasse steht, ist der zu verwendende Default-Wert `null` *("leerer Pointer")*.

e) Falls Sie das Sortieren implementiert und beibehalten wollen, schreiben Sie
"`public class LinkedList<T extends Comparable<T> >`" *(Space zwischen 2 ">"!)*
anstelle von nur
"`public class LinkedList<T>`" in der Klassen-Signatur.

   Dies dient, um sicherzustellen, dass die verwendete Klasse eine *Vergleichsoperation* bereitstellt. In Ihrem Sortieralgorithmus müssen Sie dazu passend "`if (a.data < b.data)`" nach "`if (a.data.compareTo(b.data) < 0)`" ändern.

f) In der `Test`-Klasse müssen Sie schlussendlich die Verwendung von `LinkedList` ebenfalls anpassen. Als Typ `T` übergeben wir die Klasse `Integer`. Also z. B.: `LinkedList<Integer> sll = new LinkedList<Integer>(); Integer` speichert eine Zahl wie `int` auch. Aber `int` ist ein Wertetyp und `Integer` ist eine Klasse.

g) Die Lösung von letzter Woche sollte nun identisch funktionieren wie bisher.

6.4 Generic Growing Array

Freiwillig: Wenn Sie `GrowingArray` nun auch generisch machen wollen, werden Sie über ein Problem stolpern: Java erlaubt keine Arrays von Typenparametern. Stattdessen müssen Sie einen Array der Klasse `Object` verwenden. `Object` ist eine Überklasse. Sämtliche Klassen sind automatisch auch vom Typ `Object`. Somit können wir in einem `Object`-Array sämtliche Objekte ablegen.

Damit die get-Methode trotzdem einen Wert vom Typ `T` zurückgibt, können Sie an dieser Stelle eine (unsichere) Typenkonversion verwenden: `return (T)data[ind];`

Wiederholen Sie den Performance-Vergleich von letzter Woche. Da die zu Vergleichs-Klassen selbst auch generisch sind, kämpfen Sie nun quasi mit gleich langen Spiessen. Was stellen Sie fest? Können Sie es begründen?

Viel Glück & Erfolg!