Übungsblatt 08 – Wir runden Ihre Kenntnisse in OOP ab

Abzugeben: 18.-22.12.

Wir werden Schritt für Schritt auf objektorientierte Weise eine kleine Containerbibliothek[1] entwickeln. Sie hatten ja schon einmal eine Art Container im Aufgabenblatt4 geschrieben, wo Sie nur longs mittels add hinzufügen und remove entfernen konnten. Dieses Mal schreiben wir das generisch für Objekte und wir machen das etwas effizienter. Statt bei jedem Hinzufügen eines Objekts das Feld zu erweitern, werden wir zunächst ein größeres Feld anlegen und uns merken wie viele Elemente wir gespeichert haben. Kommt jetzt ein neues Element hinzu, fügen wir das auf den nächsten freien Platz ein und erhöhen die Merkvariable um eins. Bei remove, entfernen wir das Objekt, aber verkleinern eben nicht das Feld mit Objekten wieder.

1. Zunächst starten wir mit der Basisklasse:
   a. Schreiben Sie zunächst eine Klasse MyAbstractContainer, die Sie als abstract definieren, da es keinen Sinn macht diese Klasse selbst zu instanziieren.
   b. Die Klasse soll folgende protected-Attribute besitzen: ein Feld Object [] _objs in dem Objekte verschiedener Typen gehalten werden sowie ein Attribut int _size, das die Anzahl der derzeit gehaltenen Objekte angibt.
   c. Schreiben Sie jetzt einen Konstruktor MyAbstractContainer(int capacity). In dem Konstruktor erzeugen Sie jetzt das Feld mit der Länge capacity und Sie setzen _size=0 (wir haben ja noch keine Objekt).
   d. Ferner schreiben Sie noch folgende Methoden:
      a. Public int capacity() gibt die Kapazität zurück (==die Länge des Felds).
      b. Public int size() gibt die Anzahl der derzeit gespeicherten Elemente zurück.
      c. Public Object get(int index) gibt Ihnen das Objekt an der Stelle index zurück. Falls index kein gültiger Index ist, werfen Sie eine RuntimeException mit einer sprechenden Nachricht.
      d. Public int find(Object obj) sucht das Objekt obj im Feld und gibt Ihnen den Index zurück. Falls es nicht gefunden wird, gibt find -1 zurück.
      e. Überschreiben Sie die toString()-Methode, so dass sie folgenden String zurückgibt[2]:
         <Objektname[3]> (Cap: <Capacity>, Size: <Size>):
         <Objekt1 in der Liste>[4]
         <Objekt2 in der Liste>
         …

[1] Als Container bezeichnet man ein Objekt, das mehrere andere Objekte verwaltet. Bisher haben Sie ja nur Felder als Container kennengelernt. Wie Sie aber festgestellt haben ist es sehr mühsam in ein Array ein weiteres Objekt hinzuzufügen, oder zu löschen. Deshalb werden dann in der professionellen Entwicklung üblicherweise spezielle Containerklassen verwendet, die die Verwaltung mehrerer Objekte vereinfachen.
[2] Wenn Sie es sich zutrauen, dann schreiben Sie die Methode effizient, indem Sie die Klasse StringBuilder benutzen.
[3] Den Klassennamen bekommen Sie generisch durch folgenden Befehl: this.getClass().getName();
[4] Hier können Sie einfach die toString()-Methode des entsprechenden Objekts aufrufen.

2. Wir schreiben jetzt einen Stack. Ein Stack ist auch ein Container, der allerdings nach einer "Last In, First Out"-Strategie funktioniert (Siehe auch: http://de.wikipedia.org/wiki/Stack).
   a) Schreiben Sie also eine Klasse MyStack, die von MyAbstractContainer erbt.
   b) Schreiben Sie dann einen Konstruktor MyStack(int capacity) und rufen Sie den Konstruktor der Oberklasse auf, um die Oberklasse auch richtig zu initialisieren.
   c) Schreiben Sie jetzt eine Methode public void push(Object obj), das ein Objekt auf dem Stapel hinzufügt (Objekt im Array hinten anfügen). Falls das Feld voll ist, werfen Sie eine Exception.
   d) Schreiben Sie dann eine Methode public Object pop() bei dem Sie das oberste Element aus dem Stapel entfernen (d.h. letztes Element aus dem Array entfernen) und zurückgeben.
   e) Schreiben Sie nun eine main-Methode mit der Sie MyStack testen. Sie können als Objekte z.B. die vorher von Ihnen erzeugen Klassen Wuerfel, Gladiator, Arena, Auto, Fahrrad,… verwenden. Sie sollten den Inhalt des Containers auch mind. ein Mal auf der Konsole ausgeben. Und Sie sollten auch testen, ob bei pop die Exception ausgelöst wird.

3. Wir schreiben jetzt eine einfache Listenklasse:
   a. Schreiben Sie eine Klasse MyList, die von MyAbstractContainer erbt.
   b. Schreiben Sie dann einen Konstruktor MyList(int capacity) und rufen Sie den Konstruktor der Oberklasse auf, um die Oberklasse auch richtig zu initialisieren.
   c. Schreiben Sie jetzt eine Methode public void add(Object obj), die ein Objekt in dem Array hinten anfügt und _size um eins erhöht. Falls im Feld _objs kein freier Platz mehr ist, erzeugen Sie zunächst ein neues Feld, das die doppelte Grösse von _objs hat, kopieren die alten Elemente in dieses neue Feld und lassen dann _objs auf das neue Feld zeigen[5].
   d. Schreiben Sie jetzt eine Methode public boolean remove(Object obj) in der Sie ein obj aus dem Feld _objs suchen und entfernen, wenn Sie es gefunden haben (Rückgabe: true). Falls es nicht vorhanden war geben Sie false zurück. Das Entfernen können Sie hier folgendermaßen machen: Angenommen Sie haben das zu entfernende Objekt am Index iFound gefunden. Dann verschieben Sie einfach alle Objekt mit index>iFound eine Stelle nach unten im Feld _objs (neuerIndex=index-1) und reduzieren auch _size um eins.
   e. Schreiben Sie nun eine main-Methode mit der Sie MyList testen. Sie können als Objekte z.B. die vorher von Ihnen erzeugen Klassen Wuerfel, Gladiator, Arena, Auto, Fahrrad,… verwenden. Sie sollten den Inhalt des Containers auch mind. ein Mal auf der Konsole ausgeben.

4. Schreiben Sie ein Interface IMyComparable, das eine Methode int compareTo(Object obj) definiert mit der Objekte miteinander verglichen werden können.

5. Wir schreiben jetzt eine neue Klasse für sortierte Listen:
   a. Schreiben Sie eine Klasse MySortedList, die von MyList erbt.
   MySortedList soll nur Objekte akzeptieren und beinhalten, die miteinander vergleichbar sind. Deshalb akzeptieren wir nur Objekte, die IMyComparable implementieren. Das

[5] Tipp: Schreiben Sie doch dafür eine private Methode void ensureObjsHasSpace() wo Sie überprüfen, ob _objs noch genug Platz hat und Sie es entsprechend erweitern, wenn nicht. Diese Methode können Sie dann immer ganz zu Anfang in der add-Methode aufrufen und dann haben Sie das Problem schön handlich in einer eigenen „Schublade“ gelöst und Ihre add-Methode bleibt wesentlich handlicher.

kann man erreichen, indem man vorhandene Methoden überschreibt, jedoch die Parametertypen auf den gewünschten Typ einschränkt[6].

b. Schreiben Sie eine Methode public void add(IMyComparable obj), die public void add(Object obj) überschreibt. Sie können die Methode der Oberklasse per super.add(…) aufrufen. D.h. Sie müssen die Methode nicht komplett neu entwickeln, sondern Sie definieren nur eine Methode, die die akzeptierten Typen einschränkt.
c. Schreiben Sie eine Methode IMyComparable get(int index), die Object get(int index) „überschreibt“[7]. Die Methode kann mittels super.get(index), die Methode der Oberklasse nutzen, jedoch müssen Sie dann das Ergebnis zunächst zu IMyComparable casten und dann erst das Objekt zurückgeben. Durch überschreiben der get-Methode stellen Sie sicher, dass Sie auch immer gleich ein IMyComparable bekommen. Damit sparen Sie sich hier unnötiges Casten auf IMyComparable.
d. Schreiben Sie jetzt eine Methode private void sort(), die das Feld intern sortiert (z.B. per Bubblesort). Erweitern Sie dann noch die add-Methode, indem Sie nach dem hinzufügen gleich die sort()-Methode aufrufen. Somit sind dann die Objekte immer sofort richtig sortiert.
e. Schreiben Sie nun eine main-Methode mit der Sie MySortedList testen. Sie können als Objekte z.B. die vorher von Ihnen erzeugen Klassen Wuerfel, Gladiator, Arena, Auto, Fahrrad,… verwenden[8]. Sie sollten den Inhalt des Containers auch mind. ein Mal auf der Konsole ausgeben.

6. Schreiben Sie jetzt eine MyContainerException, die von Exception erbt.
   a. Schreiben Sie einen Konstruktor public MyContainerException(String msg), der eine Nachricht (msg) annimmt und rufen Sie dann den super-Konstruktor auf, um die Ausnahmen sauber zu erzeugen.
   b. Ersetzen Sie nun in den Methoden, in denen Sie in den Aufgaben vorher Runtime-Exception verwendet haben, die RuntimeException durch eine MyContainerException. Kompilieren Sie und machen Sie die Tests erneut.

[6] Dieses funktioniert, solange der Typ der überschriebenen Methode ein Untertyp des ursprünglichen Typs ist (Hier ist IMyComparable ein Untertyp von Object). Diese Eigenschaft nennt man Kovarianz (siehe z.B. http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_05_010.htm#t2t36 Kap. 5.10.4).

[7] Hier müssen Sie die @Override-Annotation weglassen, damit der Kompiler das kompiliert. Überlegen Sie aber mal, warum das so ist und was die Konsequenz daraus ist. In der Nachbesprechung besprechen wir das dann nochmals genau.

[8] Dazu müssen Sie allerdings in den Klassen, die Sie der MySortedList mittels add hinzufügen wollen, das Interface IMyComparable implementieren (also z.B. bei Auto, wenn Sie Autos hinzufügen möchten).