Informatik II (D-ITET)
Frühjahrssemester 2015
Prof. Friedemann Mattern
Christian Beckel (beckel@inf.ethz.ch)

ETH Zürich
Institut für Pervasive Computing
Gruppe für Verteilte Systeme
http://vs.inf.ethz.ch

Übungsserie Nr. 6

Ausgabe: 1. April 2015
Abgabe: 15. April 2015

Hinweise

Für diese Serie benötigen Sie das Archiv
http://vs.inf.ethz.ch/edu/FS2015/I2/downloads/u6.zip.

1. Aufgabe: (7 Punkte) Klassen, Schnittstellen und Typumwandlungen

(1a) *(1 Punkt)* Zeichnen Sie ein Diagramm, das folgende Klassen, Schnittstellen und Vererbungsbeziehungen veranschaulicht:

```
interface A { }
abstract class B implements A { }
interface C extends A { }
class D extends B implements C { }
class E extends B { }
class F implements C { }
```

(1b) *(2 Punkte)* Für welche der obigen Typen *T* kann man mit *new T()* (z.B. *new C()*) **kein** Objekt erzeugen? Begründen Sie jeweils Ihre Antwort.

(1c) *(2 Punkte)* Gegeben die obigen Typdefinitionen, welche Zeilen aus *u6a1/StaticCasts.txt* sind gültig bzw. ungültig?

(1d) *(2 Punkte)* Gegeben die obigen Typdefinitionen, welche Zeilen aus *u6a1/DynamicCasts.txt* führen zur Laufzeit zu einer *ClassCastException*.

2. Aufgabe: (5 Punkte) Schnittstellen und Implementierungen

In den bisherigen Übungen haben Sie Codegerüste erhalten, die sie ausfüllen mussten. Der Grund dafür ist, dass die Signatur der Klassen so sein muss, wie die jeweiligen Tests es erwarten. Die Signatur ist eine Schnittstelle, welche die Klasse anbietet und der Test verwendet. In Java macht man diese Schnittstelle mit dem Schlüsselwort *interface* explizit. Das Bindeglied zwischen Anbieter und Verwender einer Schnittstelle kann eine sogenannte Fabrikmethode sein, die ein Objekt erstellt und als Referenz vom Typ der Schnittstelle zurückgibt. So kann man die Implementierung einer Schnittstelle vom Verwender verstecken und erhöht gleichzeitig die Wartbarkeit des Codes (siehe auch Folie 444 im Vorlesungsskript).

Das Interface *u6a2.IStack* spezifiziert die Schnittstelle für einen Stack, der *int*-Werte verwalten kann. Die Klasse *u6a2.ListStack* implementiert diese Schnittstelle, indem sie alle von der Schnittstelle geforderten Methoden implementiert. Die Fabrikmethode heisst *StackFactory.create* und soll ein neues *ListStack*-Objekt als *IStack*-Referenz zurück geben.

(2a) *(1 Punkt)* Implementieren sie die Fabrikmethode. Alle Tests werden nun erfolgreich bestanden.

(2b) *(2 Punkte)* Ergänzen Sie die Schnittstelle um die uns bekannte Methode *empty* inklusive Dokumentation. Nun werden sie vom Java-Compiler aufgefordert, die Methode in der Klasse *ListStack* auch zu implementieren. Tun Sie dies!

(2c) *(2 Punkte)* Ergänzen Sie die Klasse *u6a2.Tests* um einen Test, der überprüft, dass *empty* korrekt implementiert wurde.

3. Aufgabe: (10 Punkte) Polymorphie

Schnittstellen und abstrakte Klassen erlauben es, mehrere verschiedenartige Dinge einheitlich zu behandeln. Dafür gibt es mehrere Anwendungsmöglichkeiten, von denen ein paar in dieser Aufgabe behandelt werden.

(3a) *(4 Punkte)* Datencontainer wie Listen und Stacks werden generisch, wenn Sie anstatt mit *int* oder andere konkreten Typen mit *Objects* arbeiten. Die Klasse *GenericList* implementiert eine solche generische Liste und wird in den Methoden *toString*, *add* und *size* der Schnittstelle *IListUtils* verwendet. Erstellen Sie eine konkrete Implementierung *ListUtils* der Schnittstelle *IListUtils*. Schreiben Sie auch die Fabrikmethode *ListUtilsFactory.create*.

Hinweis: die Implementierungen der Methoden sind in grossen Teilen analog zu den Implementierungen aus der letzten Übungsserie. Orientieren Sie sich daran.

(3b) *(3 Punkte)* Die Klasse *GeometricObject* ist eine abstrakte Klasse für geometrische Objekte und definert die abstrakte Methode *area*. Die Klassen *Rectangle* und *Triangle* sind konkrete Klassen, die von *GeometricObject* ableiten. Daher müssen sie die Methode *area* implementieren. Tun Sie das für beide Klassen.

Die Klasse *GeometricObject* implementiert ausserdem die Schnittstelle *Comparable*, welche die Methode *smallerThan* definiert. Implementieren Sie diese Methode in der Klasse *GeometricObject*, so dass geometrische Objekte anhand ihres Flächeninhalts miteinander verglichen werden. Gehen Sie dabei davon aus, dass das übergebene *Comparable*-Objekt ein *GeometricObject* ist.

(3c) *(3 Punkte)* Alle Objekte, deren Klassen die Schnittstelle *Comparable* implementieren, können durch einen generischen Sortieralgorithmus sortiert werden. Die Schnittstelle *IListUtils* definiert eine solche Methode *sort*. Ergänzen Sie Ihre Implementierung aus Teilaufgabe *a)* um diese Methode. Gehen Sie dabei davon aus, dass alle Objekte der Liste *Comparable*-Objekte sind.

Hinweis: die Implementierung von *sort* ist in grossen Teilen analog zur Implementierung aus der letzten Übungsserie.

4. Aufgabe: (6 Punkte) Array-Listen und Generics

Analog zu dynamisch wachsenden Stacks gibt es manchmal auch Bedarf an dynamisch wachsenden Arrays. In Java gibt es hierfür einen Container namens *ArrayList*[1] (siehe auch Folie 460 im Skript). In dieser Aufgabe sollen Sie unter Verwendung von *ArrayList* einen Filter implementieren, der nur diejenigen Studenten das (nun nicht mehr vorhandene) Testat bestehen lässt, die eine Mindestanzahl an Punkten erreicht haben. Die Klasse *Student* ist bereits im Programmgerüst enthalten.

(4a) *(1 Punkt)* Erstellen Sie eine (leere) Klasse *Filter*, welche die Schnittstelle *IFilter* implementiert, und implementieren Sie die Fabrikmethode *FilterFactory.create*.

(4b) *(3 Punkte)* Implementieren Sie in der Klasse *Filter* die Methode *filterRaw* anhand der Schnittstellendokumentation. Diese Methode verwendet *ArrayList* als *raw type*, also als generische Array-Liste von *Objects*. Davon wird seit Java 1.5 abgeraten, worauf Sie der Compiler hinweist. Ignorieren Sie diesbezügliche Warnungen.

Hinweis: In den Tests sehen Sie, wie die Liste der Studenten aufgebaut ist.

(4c) *(2 Punkte)* Implementieren Sie nun die Methode *filterGeneric* anhand der Schnittstellendokumentation. Diese Methode spezifiziert auch den Typ der in der *ArrayList* gespeicherten Objekte. Diese Technik nennt sich in Java *Generics*[2] (Skript, Folie 463). Im Vergleich zu den Array-Listen aus Aufgabe *b)* gibt es drei wesentliche Unterschiede:

- Man sieht als Entwickler, was für Objekte in den Array-Listen gespeichert sind.
- Nur kompatible Objekte können zur Liste hinzugefügt werden.
- Man erhält aus der Array-Liste Referenzen des richtigen Typs und erspart sich die Umwandlung (den *cast*) der *Object*-Referenz in den richtigen Typ.

[1] `http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html`
[2] `http://docs.oracle.com/javase/tutorial/java/generics/index.html`

5. Aufgabe: (Für Fortgeschrittene) Stacks und Optimierungen

Sie haben in den vorherigen Übungsblättern zwei Varianten eines dynamisch wachsenden Stacks implementiert. Nun sollen Sie noch ein letztes mal die Implementierung verbessern.

Die Klasse *u6a5.ChunkedStack* speichert die Werte in einer Liste aus Arrays, implementiert durch die Klasse *ChunkList*. Beim Erreichen der aktuellen Kapazitätsgrenze wird ein neues Array *vorne* (!) an die Liste angehängt, so dass gleich für mehrere neue Werte Platz ist. Sollte durch das Entfernen von Werten das letzte Array vollständig ungenutzt zurück bleiben, wird es aus der Liste entfernt. So kann der Stack dynamisch wachsen und schrumpfen, ohne dass bei jedem Hinzufügen oder Entfernen von Werten ein grosser Aufwand nötig ist. Die Grösse der Arrays kann durch die Konstante *ChunkLists.chunkSize* eingestellt werden und ermöglicht einen anwendungsabhängigen Kompromiss zwischen feingranularer Speichernutzung mit häufigem Anlegen und Löschen von Objekten und grober Speichernutzung mit seltenem Zusatzaufwand.

(5a) Die Klasse *u6a5.ChunkedStack* implementiert die Schnittstelle *u6a2.IStack* aus Teilaufgabe 2. Die Methode *toString* ist schon fertig. Implementieren Sie die anderen Methoden anhand der Vorgaben der Schnittstelle. Verwenden Sie dabei die Methoden *addChunk* und *removeChunk* aus *ChunkList*.

(5b) Testen Sie Ihre Implementierung von *u6a5.ChunkedStack* mit den Tests aus *u6a2.Tests*. Passen Sie dazu die Fabrikmethode an, so dass diese nun ein *u6a5.ChunkedStack*-Objekt anlegt und zurückgibt. Lassen Sie dabei die bisherige Implementierung als Kommentar zurück.

(5c) Benutzen Sie nun die Tests aus *u6a5.Tests* für einen Performanzvergleich: Der Test misst die Laufzeiten der beiden Implementierungen für wachsende Stacks und gibt die Ergebnisse auf der Konsole aus. Ist das Verhalten von *ChunkedStack* gut oder schlecht für Stacks der getesteten Grösse? Sie können die Daten auch mit einem geeigneten Programm plotten!

(5d) Sehen Sie sich nun nochmal die Implementierung von *u6a5.ChunkList* an. Eine Methode dieser Klasse, die sehr oft aufgerufen wird, ist *size*. Was fällt Ihnen bei der Implementierung von *size* auf? Erwarten Sie, dass *size* mit zunehmender Grösse der Stacks schneller oder langsamer wird?

(5e) Überlegen Sie sich, wie Sie die Grösse der *ChunkList* effizienter verwalten können als es die derzeitige Implementierung von *size* tut - insbesondere soll nicht bei jedem Aufruf von *size* die ganze Liste durchlaufen werden! Implementieren Sie ihre Lösung und testen Sie sie mit den Tests aus *u6a2.Tests*. Führen Sie nun nochmals die Performanztests aus *u6a5.Tests* aus. Ist Ihre Implementierung nach der Veränderung schneller geworden?