Christian-Albrechts-Universität zu Kiel
Sommersemester 2000
Fortgeschrittenenpraktikum (Java)

Übung 3:

Ausgabetermin: 15. Mai 2000

Abgabe: 22. Mai 2000

Aufgabe 1: [Input/Output]

Für diese Aufgabe bietet es sich an, in das Paket `java.io` zu konsultieren.

Schreiben Sie ein Javaprogramm `CleanBytecode`, welches als Input den Namen eines Verzeichnisses nimmt, und in diesem alle Bytecode `*.class`-Dateien löscht. Falls kein Verzeichnis angegeben ist, sei das aktuelle Verzeichnis ("`.`") gemeint.

Aufgabe 2: [GUI]

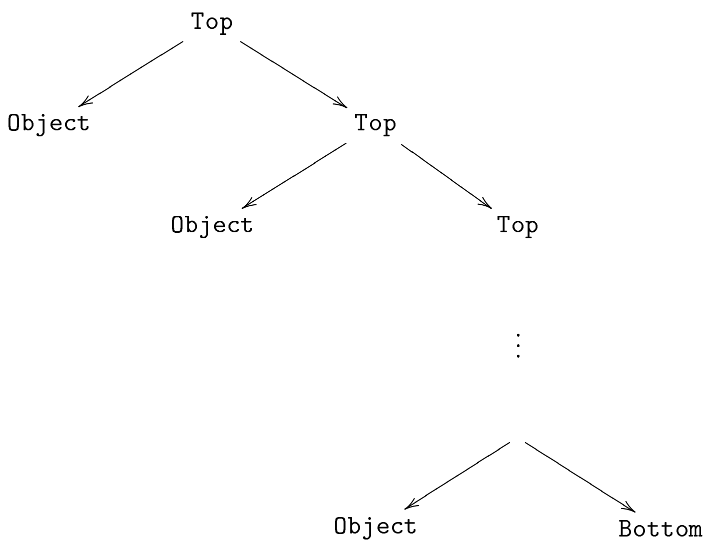
In der ersten Aufgabe der letzten Serie (Exceptions) war ein Sparkonto mit Abheben und Verzinsen gefordert, wobei die Methodenaufrufe von einer `main`-Methode aus geschahen. Peppen Sie das auf, indem Sie das Sparkonto in ein Applet einbauen. Das Applet soll zumindest die Kontonummer anzeigen und das Abheben/Einzahlen fester Beträge sowie das Anzeigen des Kontostandes mittels Knopfdruck (`Buttons`) ermöglichen. Für das Knopfdrücken können Sie sich beispielsweise am Code aus der Vorlesung oder anderen Beispielen orientieren.

Wie versprochen oder angedroht, geht es auch weiter mit den Listendatentyp. Zur Erinnerung haben wir im vorangegangenen Zettel den Schritt von einem listenförmigen "Pizzadatentyp" mit Daten wie

```
Pizza_D p =  new Paprika (new Olive (new Boden));
```

zu einer *parametrischen* Liste getan. Die Definitionen werden sich in dieser Aufgabe wieder ändern, und da es eigentlich nichts Pizza-artiges mehr hat, bezeichnen wir es gleich als Liste, aufgebaut über `Top` und `Bottom`, beides als Unterklasse von `Liste_D`. Damit bekommen konkrete Strukturen das Aussehen:[1]

[1] Eine Feinheit kann man an dieser Stelle noch anmerken: In der Listenstruktur des unteren Bildes habe ich als Elemente beliebige Objekte eingetragen. Das geht in Ordnung, um aber das *Substituieren* und *Entfernen*, wie in der alten Aufgabe verlangt, wunschgemäß implementieren zu können, müssen die Objekte, die man in die Liste hängt, auf *Gleichheit* testbar sein. Beispielsweise könnte man, wenn man die entsprechende Methode `equals` angemessen macht, in die Objekte natürlich auch wieder Listen einhängen. Um das im vorangegangenen Aufgabenzettel nicht zu kompliziert werden zu lassen, hatten wir uns auf Pizzabeläge als Objekte konzentriert.



Das war der Stand bisher. Der nächste Schritt soll sein, nicht nur (s. vorangegangene Übung 2) Pizza-Listen, bei denen man Knoblauch entfernen und Zwiebeln durch Lamm ersetzen kann, zu Listen beliebiger Dinge zu verallgemeinern, auf denen man beliebige Elemente *entfernen* und beliebige andere *ersetzen* kann (s. vorangegangene Übung 3), sondern Listen zu bekommen, bei denen man beliebige *Aktionen* ausführen kann.[2]

D.h., wenn uns neben dem Ersetzen und dem Entfernen (vertreten durch die Klassen `Remove_V` und `Substitute_V`) später noch in den Sinn kommen sollte, beispielsweise alle Elemente doppelt einzutragen, soll eine neue Klasse `Double_V` hinzugefügt werden können, *ohne* dass `Liste_D` geändert oder auch nur erweitert (im Sinne von Unterklassenbildung) werden muss.

Soweit zur Sinnfrage, nun konkreter zur neuen Aufgabe. Aus den vorangegangenen Aufgaben und aus den Lösungshinweisen geht hervor, dass die Auslagerung der Substitutions- und Entferne-Funktionalität sich in der abstrakten Listen-Klasse wie folgt niederschlägt

```
abstract class Liste_D {
  Remove_V     remove_Fn =     new Remove_V ();
  Substitute_V substitute_Fn = new Substitute_V ()

  abstract Liste_D remove(Object o);
  abstract Liste_D substitute(Object n, Object o);
};
```

und dass somit sowohl `Bottom` als auch `Top` das *Feld* mit den `remove_Fn` als Instanz des Remove-Visitors erben. Das gleiche entsprechend für das Substituieren. Um zu vermeiden, dass wir die Klasse `Liste_D` ändern müssen, sobald uns später neue Funktionalität einfällt — es kämen neue Visitor-Felder hinzu — dürfen die momentan bekannten Visitoren nicht fest in der Listen-Klasse verankert sein. Der Trick wird sein,

> die Visitoren nicht in festgelegte Felder wie `remove_Fn` oder `substitute_Fn` der Listenklasse hineinzuschreiben, sondern, ganz einfach, als *Parameter zu übergeben.*

[2] Wir werden in Aufgabe 5 genauer sehen, dass nicht wirklich wahllose neue Operationen möglich sind.

Damit es leichter wird, gehen wir in den folgenden Schritten vor.

Aufgabe 3: [Erster Schritt: Visitoren als Methodenparameter]

Wir übergeben die Visitoren (genauer natürlich ihre Instanzen) den Methoden als Parameter. Schreiben Sie zu diesem Zweck die Listenklasse um, sodass sie folgende zwei Methoden und keine Felder mehr besitzt.

```
abstract class Liste_D {
  abstract Liste_D remove(Remove_V remove_Fn, Object o);
  abstract Liste_D substitute(Substitute_V substitute_Fn, Object n, Object o);
};
```

Die Instanzen der Visitoren können dann außerhalb der Klassen definiert und als Parameter übergeben werden, zum Beispiel so:

```
Remove_V     remove_Fn     = new Remove_V();
Substitute_V substitute_Fn = new Substitute_V();


Liste_D l = new Top(new Olive(),
                    new Top(new Paprika(),
                            new Top(new Olive(),
                                    new Bottom())));


System.out.println(l.remove(remove_Fn, new Paprika()));
System.out.println(l.substitute(substitute_Fn, new Paprika()
                                               new Knoblauch());
```

Schreiben Sie die zwei Visitor-Klassen `Remove_V` und `Substitute_V`. Als Hilfestellung: die Klasse zum Entfernen hat wie vorher auch, entsprechend der Struktur unserer Listen, zwei Methoden, eine für `Bottom` eine für `Top`:

```
class Remove_V {
  Liste_D forBottom (Object o) {
    ......
  };

  Liste_D forTop (Object elem, Liste_D l, Object o) {
    ......
  };
};
```

Aufgabe 4: [zweiter Schritt: Visitoren als Konstruktorparameter]

Der zweite Schritt versucht, die im vorherigen Schritt programmierten Klassen `Remove_V` und `Substitute_V` zu vereinheitlichen. Wenn man sich die Klassen `Remove_V` und `Substitute_V` anschaut, stellt man fest, dass sie zwar die selben beiden Methoden unterstützen (`forBottom` und `forTop`), diese aber unterschiedliche *Parameterlisten* haben — die für die Substitution übergibt zwei Objekte, während die remove-Methode mit einem Objekt auskommt.

Wir können die Methodenschnittstelle *vereinheitlichen* indem wir die Objekte gar nicht an die Methodenaufrufe übergeben, sondern *den Konstruktoren* der `Remove_V`- und der `Substitute_V`-Klasse.

Schreiben Sie dazu die Visitorenklassen der vorherigen Aufgabe um, sodass man folgendes schreiben kann (`l` sei wieder eine Liste):

```
Remove_V     remove_Fn     = new Remove_V(new Paprika());
Substitute_V substitute_Fn = new Substitute_V(new Paprika(),
                                               new Knoblauch());



System.out.println(l.remove(remove_Fn));
System.out.println(l.remove(substitute_Fn));
```

Die beabsichtigte Bedeutung sollte klar sein: die Instanz `remove_Fn` entfernt Paprikas, `substitute_Fn` ersetzt Knoblauch durch Paprika.

Aufgabe 5: [Dritter Schritt: Interfaces]

Ihre beiden Visitorenklassen sollten nun *dieselbe Schnittstelle* haben; das war gerade der Sinn der vorherigen Aufgabe. In Java kann man das mittels *Interfaces* ausdrücken.

```
interface ListenVisitor {
  Liste_D forBottom () ;
  Liste_D forTop(Object elem, Liste r);
}
```

Schreiben Sie die Listen ein letztes Mal um. Die beiden Visitoren `Remove_V` und `Substitute_V` sollen das angegebene Interface *implementieren*, die abstrakte Listenklasse soll folgende abstrakte Methode enthalten, die in den zwei Unterklassen, wie zuvor bereits auch, konkretisiert werden muss:

```
abstract class Liste_D {
  abstract Liste_D accept(ListenVisitor ask);
};
```

Mit anderen Worten, die alten Methoden `remove` und `substitute` sind zusammengefasst und abstrahiert in `accept`, und das, was getan werden soll, wird der `accept`-Methode als Parameter `ask` übergeben, und wir wissen, dass es ein `ListenVisitor` ist und somit die passenden Methoden bereitstellt.

Damit soll es möglich sein, folgendes zu schreiben:

```
l.accept(new Substitute_V(new Paprika(), new Knoblauch()))
```

> Somit können wir nachträglich neue Visitoren definieren und ihre Instanzen an die Methode `accept` als Parameter übergeben.

In einer vorangegangenen Fußnote hieß es, man könne "nicht wirklich wahllos neue Operationen" der Liste nachträglich hinzufügen. Formulieren Sie, welche Operationen/Visitoren durch die `accept`-Methode möglich sind.