RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

Lehrstuhl für Hochleistungsrechnen
RWTH Aachen · D-52074 Aachen · GERMANY
http://programmierung.informatik.rwth-aachen.de

Prof. Christian Bischof, Ph.D.
Andre Vehreschild, Jakob T. Valvoda

Übung *Programmierung WS 06/07* – Blatt 6

Lösungen müssen bis zum **04. Dezember 2006, 17:00 Uhr** in den Kasten Ihrer Übungsgruppe eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55). Bitte vergessen Sie nicht, die **Nummer Ihrer Übungsgruppe**, sowie die **Namen** und **Matrikelnummern** der Mitglieder Ihrer 2er-Gruppe auf jedes Lösungsblatt Ihrer Abgabe zu schreiben. Bitte heften Sie ihre Blätter zusammen.

Alle erstellten Java-Programme sind sowohl in gedruckter Form abzugeben, als auch per E-Mail an den jeweiligen Tutor zu senden. Vergessen Sie auch bei der elektronischen Abgabe per E-Mail nicht die Angabe der **Nummer Ihrer Übungsgruppe**, sowie die jeweiligen **Namen** und **Matrikelnummern**.

Anstatt der Globalübung am **05. Dezember 2006, 15:45 Uhr** findet im **Roten Hörsaal** eine **Vorlesung** statt!

Aufgabe 1 (3 Punkte)

Beantworten Sie folgende Fragen zum Thema objekt-orientierte Programmierung und Datenabstraktion in Java:

a) Was unterscheidet eine Klasse von einem Objekt?

b) Was unterscheidet *Call-By-Value* und *Call-By-Reference*?

c) Erläutern Sie den (in der Vorlesung eingeführten) Zusammenhang zwischen Methode, Prozedur und Funktion.

d) Wie unterscheiden sich Attribute von lokalen Variablen?

e) Was ist ein ADT und durch was wird es spezifiziert? Können ADTs vollständig in Java spezifiziert werden?

f) Welche Zugriffsspezifikation sollten nichtstatische Klassenattribut in Java haben? Begründen Sie Ihre Anwort.

Aufgabe 2 (5 + 1 = 6 Punkte)

Analysieren Sie den Programmfluss des folgenden Java-Programms:

```
public class a {
  private double a = 1.0;
  private static int b = 1;
```

```
private a( int a ) {
  setA( g( new a( (double) a * 2.0 ) ) );
  setB( new a( a * 20.0 ) );
}
private a( double a ) {
  setA( (int) a * 5 );
}
private a( a a ) {
  setA( (int)a.getA() * 2 );
}
public double getA() {
  return a;
}
public void setA( int b ) {
  a = b;
}
public static void setB( a a ) {
  b = (int)a.getA();
}
public static a getInstance( int a ) {
  return new a( a );
}
public static a getInstance( double a ) {
  return new a( (int)a ) ;
}
public a getInstance( a a ) {
  return new a( a );
}
public static void f( a a ) {
  if( b < 1000 ) {
    setB( a.getInstance( a ) );
  } else {
    a.g( (double)b, a.getA() );
  }
}
public static int g( a a ) {
  return (int)( a.getA() * a.getA() );
}
public int g( int a, double b ) {
  return (int)(getInstance( a ).getA() * getInstance( b ).getA());
}
public void g( double a, double b ) {
  setB( getInstance( a ).getA() * getInstance( b ).getA() );
}

public static void main( String[] args ) {
  // Ausdruck 1
  f( getInstance( 1.0 ) );
```

```
    // Ausdruck 2
    a a = getInstance( 2 );
    // Ausdruck 3
    a.b = a.g( (int)a.getA(), 1 );
  }
}
```

a) In der Methode `main()` finden Sie drei mit Kommentaren gekennzeichnete Ausdrücke, welche jeweils mindestens einen Aufruf einer Methode der Klasse `a` beeinhalten und unter Umständen weitere Aufrufe zur Folge haben. Untersuchen Sie die Reihefolge der Methodenaufrufe und ermitteln Sie, falls möglich, die Werte der Attribute `a` und `b` *vor* Abarbeitung jeder Methode, d.h. am Anfang des entsprechenden Methodenrumpfes. Stellen Sie die Reihenfolge der Aufrufe für jeden der drei Ausdrücke nach folgendem tabellarischen Schema dar:

Methodensignatur	`a`	`b`
Signatur von `methodeA1()`	Wert von `a`	Wert von `b`
Signatur von `methodeA2()`	Wert von `a`	Wert von `b`
...	...	...
Signatur von `methodeAi()`	Wert von `a`	Wert von `b`
...	...	...

Im Falle von Ausdruck 1 sieht der Anfang der Tabelle beispielsweise folgendermaßen aus:

Ausdruck 1	`a`	`b`
`getInstance( double )`		1
Konstruktor `a( int )`	1.0	1
...	...	...

In dem dargestellten Quellcode befindet sich dabei ein Fehler, das Programm wird nicht kompilieren. Sollten Sie während der Analyse eines der drei Ausdrücke auf diesen Fehler stoßen, so beenden Sie die Auswertung des Ausdrucks, kennzeichnen Sie in der Tabelle die Methode, in der Sie den Fehler gefunden haben und fahren Sie mit dem nächsten Ausdruck fort.

b) Finden Sie den in in Aufgabenteil a) beschriebenen Fehler im Quellcode und beheben Sie diesen, ohne dabei jedoch den Programmfluß zu verändern und ohne die fehlerhafte Zeile auszukommentieren.

Aufgabe 3 (4 + 1 = 5 Punkte)

In dieser Aufgabe sollen Sie eine Klasse entwickeln, welche ein Kassensystem repräsentiert, dass in der Lage ist, mit zwei Währungen (Euro und Pfund) umzugehen (so z. B. für Flieger zwischen Euro-Europa und Grossbritanien). Das Kassensystem soll folgende Methoden zur Verfügung stellen:

- `addItem( int )` – addiert den Wert des über die angegebene Identifikationsnummer bestimmten Gegenstandes zu der Rechnung des Kunden. Die Methode überprüft selbständig, ob es sich um eine valide Identifikationsnummer handelt, ob also der Gegenstand in der Kasse gespeichert ist und somit sein Wert zu der Gesamtsumme addiert werden kann.
- `subtractItem( int )` – zieht den Wert des durch die Identifikationsnummer gegebenen Gegenstandes von der Rechnung eines Kunden ab. Hierbei ist es unwesentich, ob dieser Gegenstand auch tatsächlich früher hinzugefügt wurde.
- `float getPrice( int )` – liefert den Preis für einen Gegenstand (angegeben durch die Identifikationsnummer).
- `boolean isItem( int )` – überprüft, ob der Gegenstand (angegeben durch die Identifikationsnummer) im Kassensystem gespeichert ist, ob es sich also um eine valide Identifikationsnummer handelt.
- `addVoucher( float )` – berücksichtigt bei der Berechnung der Gesamtsumme den Wert eines Gutscheins. Es können mehrere Gutscheine eingegeben werden.
- `float getSum()` – berechnet die Gesamtsumme der Rechnung in Euro. Hierzu wird vom Gesamtwert der gekauften Artikel der Wert der eingegebenen Gutscheine abgezogen. Es werden jedoch keine Guthaben ausgezahlt, d. h. ein eventueller Restwert der Gutscheine verfällt.
- `float convertEuroToPounds( float )` – rechnet einen Euro-Betrag in Pfund um.
- `float getSumInPounds()` – berechnet die Gesamtsumme der Rechnung in Pfund.
- `float convertPoundsToEuro( float )` – rechnet einen Pfund-Betrag in Euro um.
- `reset()` – setzt die aktuelle Instanz der Klasse zurück.

Pro Kunde wird jeweils eine neue Instanz der Klasse erzeugt. Die Preise der Gegenstände werden in einem statischen `float`-Array gespeichert, wobei die Identifikationsnummer mit dem entsprechenden Array-Index übereinstimmt.

a) Implementieren Sie die Klasse `Kasse`, die obigen Methoden und den (Standard-)Konstruktor `Kasse()`, welcher alle nicht-statischen Attribute mit initialen Werten belegt. Definieren Sie hierzu die Preise für 5 Gegenstände in Ihrer Kasse. Identifizieren Sie insbesondere diejenigen Methoden, welche als `static` deklariert werden können und deklarieren Sie sie auch entsprechend als `static`! Gehen Sie von einem Wechselkurs von 1 Euro = 0,677 Pfund aus.

b) Implementieren Sie ein Testprogramm, in welchem zumindest 3 Gegenstände und ein Gutschein zu der Rechnung des Kunden hinzugefügt wird. Der Gutschein beläuft sich auf 10 Pfund. Geben Sie anschliessend die Rechnungssumme aus.

Aufgabe 4 (5 + 3 = 8 Punkte)

Analog zu dem in der Vorlesung besprochenen Laufzeitkeller, soll nun der abstrakte Datentyp Stack entwickelt werden. Der Stack soll Werte vom Datentyp `char` speichern und hat die folgenden vier Methoden

- `clear()`,
- `push( char )`,
- `char pop()` und
- `boolean isEmpty()`.

Ein Stack wird durch `clear()` gelöscht. Elemente werden mittels `push( char )` eingefügt. Das zuletzt eingefügte Element kann mit `char pop()` entfernt (und der Wert gelesen) werden. Mittels `boolean isEmpty()` kann überprüft werden, ob Elemente auf dem Stack liegen.

a) Implementieren Sie den abstrakten Datentyp Stack. Verwenden Sie hierzu ein Array, welches dynamisch an die aktuell benötigte Größe angepasst wird. Die Anpassung erfolgt wie folgt: Sobald in das Array der Länge n das $(n+1)$-te Element eingefügt werden soll, wird das Array in ein größeres Array der Länge $n + k$ kopiert und das neue Element entsprechend eingefügt. Die Vergrößerung um k Elemente erfolgt jeweils immer beim Überschreiten der aktuellen Kapazität. Wird hingegen die Anzahl der Elemente kleiner als die Schwelle $n - l$, so wird ein neues kleineres Array allociert und der Inhalt kopiert. Insgesamt soll gelten $n, k, l \in \mathbb{N}^+$. Implementieren Sie die folgenden Konstruktoren

   - `Stack( int initialSize, int increment, int decrement )`, dieser setzt initial n auf `initialSize`, sowie die Werte k auf `increment` für die Inkrementierunsschwelle und l auf `decrement` für die Dekrementierung.
   - `Stack( int initialSize )`, hierbei wird die initiale Länge n auf `initialSize` gesetzt, sowie die Werte k und l mit dem Wert 5 initialisiert.
   - `Stack()`, n wird auf 4 gesetzt. k und l werden mit 5 initialisiert.

b) Testen Sie Ihren Stack, indem Sie ihn für die Umwandlung von arithmetischen Ausdrücken aus der Infix-Notation in die Postfix-Notation (auch „umgekehrte polnische Notation“) verwenden. Während in der Infix-Notation der Operator stets zwischen den Operanden steht, wird in der Postfix-Notation der Operator nach den Operanden aufgeführt. Dadurch wird es möglich, dass Ausdrücke in Postfix-Notation ohne Klammern auskommen.

   Beispiele

   - Der Infix-Ausdruck `(5 * 6)` entspricht `5 6 *` in Postfix-Notation
   - `(5 * (6 + 7))` entspricht `5 6 7 + *`
   - `(5 * (((9 + 8) * (4 * 6)) + 7))` entspricht `5 9 8 + 4 6 * * 7 + *`

Implementieren Sie ein Programm, welches einfache arithmetische Infix-Ausdrücke in ihre Postfix-Notation umwandelt und diese ausgibt. Für Ausdrücke sind als Operanden nur Ziffern zugelassen. Desweiteren werden immer *nur zwei* Operanden von einem Operator zusammengefasst. Komplexere Ausdrücke werden durch Klammern `(` und `)` gruppiert. Es werden nur die Operatoren `+` und `*` verwendet.

Übergeben Sie den Infix-Ausdruck als Kommandozeilenargument und wandeln Sie ihn mittels `toCharArray()` in ein `char`-Array um. Beachten Sie ggf. dass Sie den Ausdruck in Anführungszeichen setzten müssen (s. letzte Übung). Nutzen Sie danach den Stack (welcher mit dem Standard-Konstruktor erstellt werden soll), um die Operatoren zwischenzuspeichern. Leerzeichen können ignoriert werden.

Testen Sie Ihr Programm, indem Sie obige Beispiele umwandeln.

Hinweis: Die Postfix-Ausdrücke sollen nur ausgegeben und nicht ausgerechnet werden!