Ludwig-Maximilians-Universität München
Institut für Informatik
PD. Dr. Peer Kröger
Johannes Niedermayer, Andreas Züfle

München, 23.01.2013

Einführung in die Programmierung

WS 2012/13

Übungsblatt 12: Typparameter, Datenstrukturen

Besprechung: 04./06./08.01.2013

Ende der Abgabefrist: Dienstag, 29.01.2013 14:00 Uhr.

Hinweise:
Die behandelten Inhalte sind Stoff der Hauptklausur, obwohl die Besprechung erst nach der Klausur stattfinden wird. Deshalb wird die Lösung zu diesem Übungsblatt am Dienstag, den 29.01.2013 nach Ablauf der Abgabefrist online gestellt. Beachten Sie, dass die Besprechung des Blattes zeitlich verschoben stattfindet, also von Montag, 04.02.20013 bis Freitag 08.02.2013.

Aufgabe 12-1 *Typparameter* **0 Punkte**

Die Klasse `MinMax` enthält zwei Methoden, die das Maximum bzw. das Minimum aus einer Menge von `Comparable`-Objekten bestimmen. Die Schreibweise `Comparable... cs` in der Parameter-Liste steht für ein oder mehrere Objekte vom Typ `Comparable`. Man kann also auf diese Weise eine Methode definieren, die eine beliebige, nicht-leere Menge von Objekten eines bestimmten Typs als Parameter erhält. Diese Schreibweise kann allerdings nur beim letzten Parameter einer Methode auftreten. Im Methodenrumpf kann der entsprechende Parameter behandelt werden wie ein Array, er kann also auch in Schleifen verwendet werden. Hier wird eine erweiterte Form der **`for`**-Schleife verwendet, die nützlich ist, wenn ein Array lediglich durchlaufen werden soll, ohne dass eine Verwendung des Schleifenindex nötig ist.

Die `main`-Methode enthält nach zwei richtigen Verwendungen der Methoden (für `Integer`- und `String`-Objekte) auch eine Verwendung, in der beide Objekt-Typen gemischt vorkommen. Dieser Fehler wird aber nicht beim Übersetzen erkannt. Korrigieren Sie die Klasse `MinMax`, so dass eine solche gemischte Verwendung bereits beim Übersetzen als Fehler erkannt wird. Damit das korrigierte Programm übersetzbar ist, müssen Sie dann natürlich die letzte Anweisung in der `main`-Methode auskommentieren. Begründen Sie Ihre Korrektur der Klasse.

Lösungsvorschlag:

Die Methoden `maximum` und `minimum` verwenden mehrmals den Typ `Comparable`.
Das Interface `Comparable` hat aber einen Typparameter, hier wird der *raw-type* verwendet. Alle Stellen in der Klasse `MinMax`, an denen `Comparable` vorkommt, werden implizit zu `Comparable<Object>` ergänzt (um die Rückwärtskompatibilität von Java zu gewährleisten).
Die formalen Parameter der Methode `maximum` haben also alle den Typ `Comparable<Object>`. Jedes Objekt `x` von `Comparable<Object>` (oder einer Unterklasse davon) hat also eine Methode `compareTo(Object)`. Das erlaubt zum Beispiel, dass `x` ein `Integer` ist und `x.compareTo` auf ein `String`-Objekt angewendet wird. Was wir eigentlich wollen, ist, dasss `x.compareTo` auf ein Objekt angewendet wird, das den selben Typ hat wie `x`. Nennen wir diesen Typ einstweilen `T`. Was wissen wir über `T`?
`T` muss eine Klasse sein, die das Comparable-Interface implementiert. Aber nicht `Comparable<Object>` (sonst könnte `x.compareTo` z.B. auf ein `String`-Objekt angewendet werden, egal, welcher Typ `T` ist), sondern `Comparable<T>`.

Das ist der Fall, wenn `T` eine Klasse ist, die `Comparable<T>` implementiert.
Mit den Sprachmitteln zur Formulierung von Typparametern wird das durch eine **`extends`**-Klausel spezifiziert: `<T extends Comparable<T>>`.
Die nächste Frage ist, wo man den Typparameter `<T extends Comparable<T>>` in der Klasse `MinMax` einführt.

Möglichkeit 1: Wir machen daraus einen formalen Typparameter der Klasse.

```
class MinMax<T extends Comparable<T>>
{
  public T maximum(T c, T... cs)
  {
    for(T x : cs)
    {
      if(x.compareTo(c)>0)
      {
        ...
```

Dadurch hat aber auch der Konstruktor den Typparameter, der bei der Objekt-Erzeugung angegeben werden muss. Dann muss man in der `main`-Methode zwei Objekte erzeugen:

```
MinMax<Integer> minMaxInteger = new MinMax<Integer>();
MinMax<String> minMaxString = new MinMax<String>();
```

`minMaxInteger.maximum` kann nur `Integer`-Objekte als Parameter bekommen.
`minMaxString.maximum` kann nur `String`-Objekte als Parameter bekommen.
Verstöße werden vom Compiler erkannt.

Möglichkeit 2: Wir können daraus einen formalen Typparameter der Methoden machen, da `T` in `MinMax` nicht außerhalb der Methoden vorkommt. Die `main`-Methode bleibt unverändert, da `MinMax` und sein Konstruktor jetzt gar keinen Typparameter haben.

```
public class MinMax {
    public <T extends Comparable<T>> T maximum(T c, T... cs) {
        for (T x : cs) {
            if (x.compareTo(c) > 0) {
                c = x;
            }
        }
        return c;
    }
    public <T extends Comparable<T>> T minimum(T c, T... cs) {
        for (T x : cs) {
            if (x.compareTo(c) < 0) {
                c = x;
            }
        }
        return c;
    }
    private static void print(String ueberschrift, Object min, Object max,
            Object... os) {
        StringBuilder builder = new StringBuilder();
        builder.append(ueberschrift);
        builder.append("\n  Elemente:\n");
        for (Object o : os) {
            builder.append("    ");
            builder.append(o.toString());
            builder.append("\n");
        }
        builder.append("  Minimum: ");
        builder.append(min);
        builder.append("\n  Maximum: ");
        builder.append(max);
        builder.append("\n");
        System.out.println(builder);
    }
    public static void main(String[] args) {
        MinMax minMax = new MinMax();
        Integer i1 = new Integer(1);
        Integer i2 = new Integer(2);
        Integer i3 = new Integer(3);
        Integer i4 = new Integer(4);
        String s1 = "ab";
        String s2 = "ac";
        String s3 = "bc";
        print("Integer", minMax.minimum(i1, i2, i3, i4), minMax.maximum(i1, i2,
                i3, i4), i1, i2, i3, i4);
        print("Integer", minMax.minimum(s1, s2, s3),
                minMax.maximum(s1, s2, s3), s1, s2, s3);
        // Fehlerhafte Verwendung: jetzt Fehler zur Uebersetzungszeit
        // print("Integer",
        // minMax.minimum(i1, i2, s3, s2),
        // minMax.maximum(i1, i2, s3, s2),
        // i1, i2, s3, s2);
    }
}
```

Aufgabe 12-2 *Datenstrukturen* **3+4+5 Punkte**

Gegeben folgende Klasse zur Modellierung einer sortierten Liste:

```
public class SortierteListe<E extends Comparable<E>> {
  private int size;
  private Entry<E> head;

  public SortierteListe() {
    this.head = null;
  }

  public E get(int index) {
    if(index < 0 || index >= this.size) {
      throw new IndexOutOfBoundsException("Index: " + index
                                          + ", Size: " + this.size);
    }
    Entry<E> currentEntry = this.head;
    while(index > 0) {
      currentEntry = currentEntry.getNext();
      index--;
    }
    return currentEntry.getElement();
  }

  public int size() {
    return this.size;
  }

  private static class Entry<E> {
    private E element;
    private Entry<E> next;

    public Entry(E o, Entry<E> next) {
      this.element = o;
      this.next = next;
    }

    public E getElement() {
      return this.element;
    }

    public void setElement(E element) {
      this.element = element;
    }

    public Entry<E> getNext() {
      return this.next;
    }

    public void setNext(Entry<E> next) {
      this.next = next;
    }
  }
}
```

Die Liste ist aufsteigend gemäß der natürlichen Ordnung ihrer Elemente sortiert, d.h. für zwei Elemente der Liste `a` und `b` gilt: Aus `a.compareTo(b) < 0` folgt, dass das Element `a` in der Liste vor dem Element `b` steht.

Ergänzen Sie diese Klasse um folgende Methoden, wobei Sie die Eigenschaft der Sortierung nutzen, um

unnötige Operationen zu sparen:

(a) Die Methode **public int** `indexOf(E element)` gibt den Index des ersten Vorkommens des gegebenen Elementes in der Liste zurück (beginnend mit 0 für das erste Element der Liste) oder -1, falls das Element in der Liste nicht vorkommt.

Lösungsvorschlag:

```
public int indexOf(E element) {
  Entry<E> currentEntry = this.head;
  int index = 0;
  while(currentEntry != null
    && currentEntry.getElement().compareTo(element) <= 0) {
    if(currentEntry.getElement().compareTo(element) == 0) {
      return index;
    }
    currentEntry = currentEntry.getNext();
    index++;
  }
  return -1;
}
```

(b) Die Methode **public void** `add(E element)` fügt der Liste das gegebene Element hinzu, wobei die Eigenschaft der Sortierung erhalten bleiben soll.

Lösungsvorschlag:

```
public void add(E element) {
  Entry<E> currentEntry = this.head;
  if(currentEntry == null) {
    this.head = new Entry<E>(element, null);
    this.size++;
    return;
  }
  if(currentEntry.getElement().compareTo(element) > 0) {
    this.head = new Entry<E>(element, currentEntry);
    this.size++;
    return;
  }
  while(currentEntry.getNext() != null
     && currentEntry.getNext().getElement().compareTo(element) < 0) {
    currentEntry = currentEntry.getNext();
  }
  Entry<E> entry = new Entry<E>(element, currentEntry.getNext());
  currentEntry.setNext(entry);
  this.size++;
}
```

(c) Die Methode **public void** `slice(E min, E max)` entfernt aus der Liste alle Elemente, die kleiner als `min` oder größer als `max` sind.

Lösungsvorschlag:

```
public void slice(E min, E max) {
  if(this.head == null) {
    return;
  }
  Entry<E> currentEntry = this.head;
  this.head = null;
```

```
      this.size = 0;
      while(currentEntry != null
         && currentEntry.getElement().compareTo(min) < 0) {
        currentEntry = currentEntry.getNext();
      }
      if(currentEntry.getElement().compareTo(max) <= 0) {
        this.head = currentEntry;
        this.size++;
      }
      while(currentEntry.getNext() != null
         && currentEntry.getNext().getElement().compareTo(max) <= 0) {
        currentEntry = currentEntry.getNext();
        this.size++;
      }
      if(currentEntry != null) {
        currentEntry.setNext(null);
      }
    }
```

Aufgabe 12-3 *Generics* **0 Punkte**

Gegeben sind folgende Klassen zur Modellierung von Krapfen mit Füllung:

```
1  public class Geschmacksrichtung {
2    public static final String[] GESCHMACKSRICHTUNGEN =
3                  {"Erdbeere", "Himbeere", "Hagebutte"};
4
5    private String geschmacksrichtung;
6
7    public Geschmacksrichtung(int geschmacksrichtung){
8      if(geschmacksrichtung < 0
9      || geschmacksrichtung >= GESCHMACKSRICHTUNGEN.length){
10       throw new IllegalArgumentException("Ungueltige Geschmacksrichtung");
11     }
12     this.geschmacksrichtung = GESCHMACKSRICHTUNGEN[geschmacksrichtung];
13   }
14
15   public String getGeschmacksrichtung(){
16     return this.geschmacksrichtung;
17   }
18 }
```

```
1  public class Fuellung {}
```

```
1  public class Marmelade extends Fuellung {
2    private Geschmacksrichtung geschmacksrichtung;
3
4    public Marmelade(Geschmacksrichtung geschmacksrichtung){
5      this.geschmacksrichtung = geschmacksrichtung;
6    }
7
8    public Geschmacksrichtung getGeschmacksrichtung(){
9      return this.geschmacksrichtung;
10   }
11 }
```

```java
public class Senf extends Fuellung {}
```

```java
public class Krapfen {
  private Fuellung fuellung;

  public void fuellen(Fuellung fuellung){
    this.fuellung = fuellung;
  }

  public Fuellung schmecktNach(){
    return this.fuellung;
  }
}
```

Diese Modellierung wird von einer Reihe von Klassen verwendet:

```java
import java.util.ArrayList;
import java.util.List;

public class Konditor {
  public List<Krapfen> krapfenMachen(int anzahl, Fuellung fuellung)
  {
    List<Krapfen> krapfenliste = new ArrayList<Krapfen>(anzahl);
    for(int i = 0; i < anzahl; i++)
    {
      Krapfen krapfen = new Krapfen();
      krapfen.fuellen(fuellung);
      krapfenliste.add(krapfen);
    }
    return krapfenliste;
  }
}
```

```java
public class Kunde {
  public Geschmacksrichtung krapfenEssen(Krapfen krapfen){
    return ((Marmelade) krapfen.schmecktNach()).getGeschmacksrichtung();
  }
}
```

```java
import java.util.List;

public class Krapfenkauf{
  public static void main(String[] args){
    Konditor grattler = new Konditor();
    List<Krapfen> krapfenliste = grattler.krapfenMachen(10, new Senf());
    Kunde kunde = new Kunde();
    for(Krapfen krapfen : krapfenliste)
    {
      if(kunde.krapfenEssen(krapfen)
              .getGeschmacksrichtung()
              .equalsIgnoreCase("ERDBEERE"))
      {
        System.out.println("Mmmmmh!");
      }
    }
  }
}
```

(a) Bei Ablauf des Programmes `Krapfenkauf` wird ein Fehler auftreten. Um was für einen Fehler handelt es sich, warum und an welcher Stelle (Klassenname, Zeilennummer) tritt er auf?

Lösungsvorschlag:

In `Krapfenkauf`, Zeile 11 wird die Methode `Kunde.krapfenessen()` aufgerufen, in der die `Fuellung` nach `Marmelade` gecastet wird. Da es sich diesmal bei der `Fuellung` um `Senf` handelt, führt diese Methode in Zeile 5 zu einer `ClassCastException`.

(b) Verbessern Sie die Modellierung von Krapfen mit Füllung durch Typisierung so, dass bei Ihrer Verwendung Typsicherheit zur Übersetzungszeit erreicht wird.

(Es genügt, wenn Sie hier nur die geänderten Zeilen unter Angabe von Klassenname und Zeilennummer eintragen.)

Lösungsvorschlag:

Zu ändern ist lediglich die Klasse `Krapfen` selbst:

```
public class Krapfen<T extends Fuellung> {

  private T fuellung;

  public void fuellen(T fuellung){
    this.fuellung = fuellung;
  }

  public T schmecktNach(){
    return this.fuellung;
  }
}
```

(c) Passen Sie die Klassen `Konditor`, `Kunde` und `Krapfenkauf` dieser neuen Modellierung an. Ein Kunde soll danach grundsätzlich nur Krapfen essen, die mit Marmelade gefüllt sind. Ansonsten soll sich aber im Ablauf und Ergebnis des Programmes nichts ändern.

(Es genügt, wenn Sie hier nur die geänderten Zeilen unter Angabe von Klassenname und Zeilennummer eintragen.)

Lösungsvorschlag:

```
public class Konditor {

  public <T extends Fuellung> List<Krapfen<T>> krapfenMachen(int anzahl,
                                                             T fuellung)
  {
    List<Krapfen<T>> krapfenliste = new ArrayList<Krapfen<T>>(anzahl);
    for(int i = 0; i < anzahl; i++)
```

```
    {
      Krapfen<T> krapfen = new Krapfen<T>();
      krapfen.fuellen(fuellung);
      krapfenliste.add(krapfen);
    }
    return krapfenliste;
  }

}

public class Kunde {

  public Geschmacksrichtung krapfenEssen(Krapfen<Marmelade> krapfen)
  {
    return krapfen.schmecktNach().getGeschmacksrichtung();
  }

}

public class Krapfenkauf {

  public static void main(String[] args) {
    Konditor grattler = new Konditor();
    List<Krapfen<Marmelade>> krapfenliste = grattler.krapfenMachen(10,
                                                                   new Senf());
    Kunde kunde = new Kunde();
    for(Krapfen<Marmelade> krapfen : krapfenliste)
    {
      if(kunde.krapfenEssen(krapfen)
              .getGeschmacksrichtung()
              .equalsIgnoreCase("ERDBEERE"))
      {
        System.out.println("Mmmmmh!");
      }
    }
  }
}
```

(d) Der Fehler aus Teilaufgabe (a) bleibt grundsätzlich bestehen, es ist aber nun ein Fehler anderer Art. Um welche Art Fehler handelt es sich jetzt, wo und warum tritt er auf? Was ist dennoch mit der geänderten Modellierung gewonnen?

Lösungsvorschlag:

In der Musterlösung tritt nun ein type mismatch in Zeile 5 auf, da der Konditor grattler seine Krapfen mit Senf füllen will, das Ergebnis aber mit Marmelade gefüllte Krapfen sein sollen. (Je nach Lösung kann sich das Auftreten des Fehlers auch verschieben.)

Im Gegensatz zur `ClassCastException` ist dies aber ein Fehler zur Übersetzungszeit und wird damit bereits vor Gebrauch der Klasse erkannt.