Prof. Dr. Stefan Brass
M.Sc. Mario Wenzel

Martin-Luther-Universität Halle-Wittenberg
Institut für Informatik
Wintersemester 2018/19

Objektorientierte Programmierung: Hausaufgabenblatt 13

Abgabe: 04.02.2019, 11:00

Das Ziel dieses Übungsblattes ist es, dass Sie ihre Kenntnisse zu Klassenhierarchien und Interfaces festigen.
Diese Übungsserie finden Sie im Stud.IP auf der Seite der Veranstaltung unter
Lernobjekte → Kurs in ILIAS → Hausaufgabenblatt 13.

Hausaufgabe 1: (4 Punkte)

Bitte beachten Sie: die Einsendung der Aufgabe erfolgt über die ILIAS-Plattform.

Gegeben sei folgende Klassenhierarchie:

```
class X {}
class Y extends X {}

abstract class P {
        abstract X f(X i);
}
```

Welche der folgenden Klassen A, B, C, D erweitert die Klasse P korrekt um eine Implementierung für die Methode f? Wenn eine Klasse das nicht tut, geben Sie den Grund an (Mehrfachnennung möglich).

```
class A extends P {
  private X f(X i) { return i; }
}

class B extends P {
  Y f(X i) { return new Y(); }
}
```

```
class C extends P {
  X f(Y i) { return new X(); }
}

class D extends P {
  public X f(X z) { return z; }
}
```

	A	B	C	D
Ja, korrekt erweitert				
Nein, denn die Argumenttypen unterscheiden sich				
Nein, denn der Rückgabetyp unterscheidet sich				
Nein, denn die Argumentnamen unterscheiden sich				
Nein, denn die Sichtbarkeit ist eingeschränkt				
Nein, denn die Sichtbarkeit ist erweitert				

Hausaufgabe 2: (6 Punkte)

Bitte beachten Sie: die Einsendung der Aufgabe erfolgt nur im YAPEX.
Open exercises via code → Freigabecode: `5d0fci753e2d-0398` (Term Interface)

Interfaces sind nützlich bei der Betrachtung von rekursiv definierten (mathematischen) Strukturen. Dies kennen Sie bereits aus der Schulmathematik. So ist Beispielsweise ein Term definiert als (1) eine Konstante, (2) eine Variable oder (3) eine Operation, die als jeden Operanden einen Term (rekursive Definition) hat.
Gleichwohl könnten wir die Natürlichen Zahlen definieren als (1) Null oder (2) der Nachfolger einer natürlichen Zahl. Dies ist eine nützliche Betrachtungsweise.
Das Interface passt zur Struktur selbst (Term, Natürliche Zahl) und jede “Regel” besitzt eine Klasse, die dieses Interface implementiert.

Gegeben Sei das folgende Interface für Terme mit Werten aus den ganzen Zahlen:

```
interface Term {
  int value();
  int depth();
  String toString();
}
```

Schreiben Sie die Klassen `Number`, `Plus`, `Times` die allesamt das `Term`-Interface implementieren. Die Funktionalität der einzelnen Klassen soll wie folgt sein:

- Parameter der Konstruktoren:
  - `Number`: `int`
  - `Plus`, `Times`: `Term, Term` (linke Seite, rechte Seite)
- `value()` - der Wert des Terms
  - `Number`: die Zahl selbst
  - `Plus`: Summe der Werte der rechten und linken Seite
  - `Times`: Produkt der Werte der rechten und linken Seite
- `depth()` - die Termtiefe (wie viele Ebenen hat die Baumstruktur)
  - `Number`: 0
  - `Plus, Times`: Maximum der Tiefen der rechten und linken Seite erhöht um 1
- `toString()`
  - `Number`: die Zahl selbst
  - `Plus, Times`: Die String-Darstellung der Rechten Seite und der Linken Seite mit dem Verknüpfungszeichen in der Mitte (ohne Leerzeichen), der gesamte Ausdruck in Klammern

Ihre Klassen werden automatisch von YAPEX getestet. Sie brauchen keine Main-Funktion zu schreiben! Dabei prüfen wir unter Anderem folgende Operationen, die den Wert (im Kommentar) ergeben sollen

```
Term a = new Number(5);
a.value(); // 5
a.depth(); // 0
Term b = new Plus(a, new Number(3));
b.value(); // 8
Term c = new Times(a, b);
Term d = new Times(c, c);
c.value(); // 40
d.value(); // 1600
d.depth(); // 3
c.toString(); // "(5*(5+3))"
```

Hinweis: Wenn Sie für die Termtiefe die Funktion `max` aus der `Math`-Bibliothek benutzen, können Sie alle Methoden (nicht den Konstruktor) in je einer Zeile (mit einer Anweisung) implementieren. Sie benötigen auch keine weiteren Hilfsmethoden oder weitere Attribute, bis auf die zum Abspeichern der den Konstruktoren übergebenen Werte (die Attribute können alle final und private sein).

Weitere Betrachtungen: Aus Sicht der Programmstrukturierung könnte man nun leicht weitere Klassen schreiben, die das `Term`-Interface implementieren (für weitere Operatoren). Für die Umsetzung von Variablen müsste man jedoch die `value`-Funktion um Informationen zur Variablenbindung erweitern, da sonst nicht klar ist, was der Wert für `(new Variable("X")).value()` sein soll (jede Methode des Interface muss für alle das Interface implementierenden Klassen einen sinnvollen Wert ergeben). Als weitere Übung könnten Sie jedoch versuchen die unäre Operation “Minus” bzw. “Negation” aufzuschreiben.

Hausaufgabe 3: (4 Bonuspunkte — Abgabe bis 10.02.2019)

Bitte beachten Sie: die Einsendung der Aufgabe erfolgt über die ILIAS-Plattform.

Bisher wurden immer nur konkrete Klassen betrachtet und konstruiert und es wurde nur über die *Werte* von Argumenten und Rückgabe parametrisiert (der konkrete Typ der Parameter wurde beim Programmieren festgelegt, die Werte erst später berechnet). Bei der generischen Programmierung wird auch über Typen parametrisiert. Sie stehen beim Programmieren noch nicht fest und werden erst später konkret instanziiert. Eine Klasse oder ein Interface hat dann einen (oder mehrere) sog. Typparameter der beim Instanziieren einer konkreten Klasse festgelegt wird.

Gegeben sei ein Ausschnitt des Interface `List⟨E⟩` mit der konkreten Implementation `LinkedList⟨E⟩` mit Typparameter `E`, sowie die Typhierarchie A,B.

```
public interface List<E> extends Collection<E> {
  boolean add(E o);
  E get(int index);
}

class A {}
class B extends A {}
```

Gehen Sie davon aus, dass `LinkedList⟨E⟩` korrekt `List⟨E⟩` implementiert (nicht gezeigt). Geben Sie an, welche der folgenden Anweisungen (statisch) erlaubt sind (ignorieren Sie mögliche Laufzeitfehler beim Aufruf der `get`-Methode):

Anweisung	Erlaubt	Nicht erlaubt
`List⟨A⟩ l1 = new LinkedList⟨A⟩();`		
`List⟨A⟩ l2 = new LinkedList⟨B⟩();`		
`List⟨B⟩ l3 = new LinkedList⟨A⟩();`		
`(new LinkedList⟨A⟩()).add(new A());`		
`(new LinkedList⟨A⟩()).add(new B());`		
`(new LinkedList⟨B⟩()).add(new A());`		
`A v1 = (new LinkedList⟨A⟩()).get(0);`		
`A v2 = (new LinkedList⟨B⟩()).get(0);`		
`B v3 = (new LinkedList⟨A⟩()).get(0);`		

Weitere Betrachtung: Man könnte das Term-Interface auch um einen Typparameter für den Werttyp erweitern (bisher war dieser auf `int` fixiert). Dafür muss das Auftreten von `int` für den Wert durch den Typparameter ersetzt werden. Die möglichen Operationen muss man dann aber wahrscheinlich für konkrete Typen implementieren.