



Übung zur Vorlesung *Einführung in die Informatik 2 für Ingenieure (MSE)*

Alexander van Renen (renen@in.tum.de)

<http://db.in.tum.de/teaching/ss16/ei2/>

Lösungen zu Blatt 3

Aufgabe 1: Overloading

Welche der folgenden Methoden-Überladungen sind erlaubt und welche nicht? Überprüfen Sie Ihre Antworten indem Sie die Beispiele in Java programmieren.

Lösung 1

Das Laufzeitsystem muss herausfinden, welche der beiden überladenen Methoden es aufrufen muss, wenn diese im laufenden Programm verwendet werden. Damit dies möglich ist, müssen sich diese entweder in der Anzahl der Parameter oder in den Parametertypen unterscheiden.

- a) **Nicht erlaubt:** Beide Methoden haben genau einen Parameter vom Typ `double` und sind damit nicht unterscheidbar.

Car
-speed : double
+accelerate(mph : double)
+accelerate(kmh : double)

- b) **Erlaubt:** Hier gibt es keinen Konflikt, da die Methoden in verschiedenen Klassen definiert sind.

MetricCar
-speed : double
+accelerate(kmh : double)

ImperialCar
-speed : double
+accelerate(mph : double)

- c) **Erlaubt:** Die verschiedene Parametertypen erlauben dem Laufzeitsystem die Unterscheidung.

Car
-speed : double
+accelerate(kmh : double)
+accelerate(seconds : int)

- d) **Nicht erlaubt:** Der Typ des Rückgabewertes wird nicht zur Unterscheidung herangezogen.

Car
-speed : double
+accelerate(seconds : int)
+accelerate(seconds : int) : double

- e) **Nicht erlaubt:** Klassenmethoden werden in diesem Fall wie Objektmethoden behandelt.

Car
-speed : double
+toString() : String
<u>+toString() : String</u>

f) **Erlaubt:** Die verschiedene Anzahl Parameter ermöglicht die Unterscheidung.

Car
-speed : double
+accelerate(kmh : double)
+accelerate(mph : double, slope : double)

g) **Nicht erlaubt:** Die generische Typinformation wird bei der Kompilierung entfernt und steht zur Laufzeit nicht mehr zu Verfügung. Beide Methoden haben dadurch genau einen Parameter vom Typ `Set<Object>` sind damit nicht mehr unterscheidbar.

Car
-speed : double
+load(passengers : Set<Passenger>)
+load(luggage : Set<Suitcase>)

Aufgabe 2: Dynamisches Binden

Überlegen Sie sich welche Methoden aufgerufen werden, wenn man die Klasse `DynamicDispatch` ausführt. Überprüfen Sie anschließend Ihre Vermutung, indem Sie das Programm tatsächlich ausführen.

```

1 class DynamicDispatch {
2     public static void main(String [] args) {
3         A a = new A();
4         B b = new B();
5         C c = new C();
6         D d = new D();
7
8         A[] array = {a, b, c, d};
9         for (A element : array) {
10            System.out.println("x() :");
11            element.x();
12            System.out.println("\ny() :");
13            element.y();
14            System.out.println("\nz() :");
15            element.z();
16            System.out.println("\n=====\n");
17        }
18    }
19 }
20
21 class A {
22     public void x() {

```

¹Unterstrichen bedeutet hier Klassenmethode.

```

23     System.out.println ("->A_x() ");
24     z();
25 }
26
27 public void y() {
28     System.out.println ("->A_y() ");
29     this.z();
30 }
31
32 public void z() {
33     System.out.println ("->A_z() ");
34 }
35 }
36
37 class B extends A {
38     public void y() {
39         System.out.println ("->B_y() ");
40         x();
41     }
42
43     public void z() {
44         System.out.println ("->B_z() ");
45     }
46 }
47
48 class C extends B {
49     public void x() {
50         System.out.println ("->C_x() ");
51         z();
52     }
53 }
54
55 class D extends A {
56     public void x() {
57         System.out.println ("->D_x() ");
58         super.x();
59     }
60
61     public void z() {
62         System.out.println ("->D_z() ");
63     }
64 }

```

Lösung 2

Das Konzept der Vererbung beinhaltet in Java, dass eine Variable von einem bestimmten Klassentyp auch Objekte von deren Untertklassen referenzieren kann. Ruft man nun eine Methode auf dem durch die Variable referenzierten Objekt auf, gibt es zwei Möglichkeiten:

- Führe die Methode der Oberklasse aus
- Führe die überschriebene Methode einer der Unterklassen aus

Von *dynamischer Bindung* spricht man, wenn in solchen Fällen immer die speziellste Variante der Methode aufgerufen wird, d.h. bei einem Objekt der Unterklasse würde, wenn diese existiert, die überschriebene Methode dieser Unterklasse aufgerufen werden. Wird stattdessen immer die Methode der Oberklasse genutzt, so spricht man von *statischer Bindung*. Bei Java hat man dabei keine Wahl: Methoden werden immer dynamisch gebunden. Die dynamische Bindung ist von enormer Bedeutung für die objektorientierte Programmierung, da die Flexibilität der Vererbung nur durch dynamische Bindung zum Tragen kommt. Ohne dynamische Bindung könnten Unterklassen das Verhalten der Oberklasse nicht anpassen, sondern nur um neue Methoden erweitern.

Zurück zur eigentlichen Aufgabe: Die Ausgabe des Programms verdeutlicht, in welcher Reihenfolge die Methoden aufgerufen werden.

Aufgabe 3: Don't repeat yourself (DRY)

Auf der Website² finden Sie die Klasse `Warenhaus.java`.

Lösung 3

Redundant vorhandene Informationen (z. B. Code-Duplizierungen in Quellcode) sind schwierig zu pflegen, da die Konsistenz zwischen den einzelnen Duplikaten von Hand gewährleistet werden muss. Vergisst man bei Änderungen eine Stelle, kommt es zu Inkonsistenz und als Folge oft zu nur schwer nachvollziehbarem Fehlverhalten des Programms. In dem Programm aus diese Aufgabe ist dies die Preisberechnung mit der Mehrwertsteuer, die an vielen Stellen redundant vorkommt. Bei Systemen, die dem DRY-Prinzip treu bleiben, brauchen Änderungen hingegen nur an einer Stelle vorgenommen zu werden und es können dabei keine Inkonsistenzen entstehen.

a) Nennen Sie mindestens zwei Gründe, warum die Modellierung nicht optimal ist – beispielsweise, wenn sich der Mehrwertsteuersatz mal wieder ändern sollte.

- Die Mehrwertsteuer ist in den Klassen mehrfach fest eingetragen. Diese Redundanz kann zu Inkonsistenzen führen, da bei einer Erhöhung des Steuersatzes eine Stelle übersehen werden könnte. Der Steuersatz sollte stattdessen an einer einzigen Stelle definiert sein und von dort jeweils referenziert werden.
- Alle fünf Produkte haben eine unterschiedlich benannte Methode um ihren Preis festzustellen. Dies führt zu Mehraufwand bei der Verwendung der Klassen, da ein Klient erst herausfinden muss, wie die jeweilige Methode heißt.
- Die fünf Produktklassen haben keine gemeinsame Oberklasse, obwohl sie ähnliches Verhalten aufweisen. Eine gemeinsame Oberklasse würde es erlauben, Objekte der verschiedenen Klassen gleich zu behandeln – beispielsweise könnten diese dann in Kollektionen zusammengefasst werden.

b) Implementieren Sie den Aufzählungstyp `Mehrwertsteuersatz`, der zwei Werte für den normalen und den vergünstigten Mehrwertsteuersatz hat.

Auf der Website finden Sie die Lösung `WarenhausLoesung.java`.

²Hier: <http://www3.in.tum.de/teaching/ss13/ei2/>

- c) Ziehen Sie die Gemeinsamkeiten der Produkte in eine gemeinsame Oberklasse **Produkt**. Führen Sie eine Methode ein, die für ein Produkt angibt, ob die vergünstigte Mehrwertsteuer anwendbar ist. Verwenden Sie diese Methode bei der Preisberechnung.

Auf der Website finden Sie die Lösung `WarenhausLoesung.java`.

- d) Führen Sie die Klasse **Einkaufskorb** ein, die eine Menge von Produkten verwaltet und deren Gesamtpreis bestimmen kann. Verwenden Sie diese Klasse in der main-Methode des Warenhauses.

Auf der Website finden Sie die Lösung `WarenhausLoesung.java`.