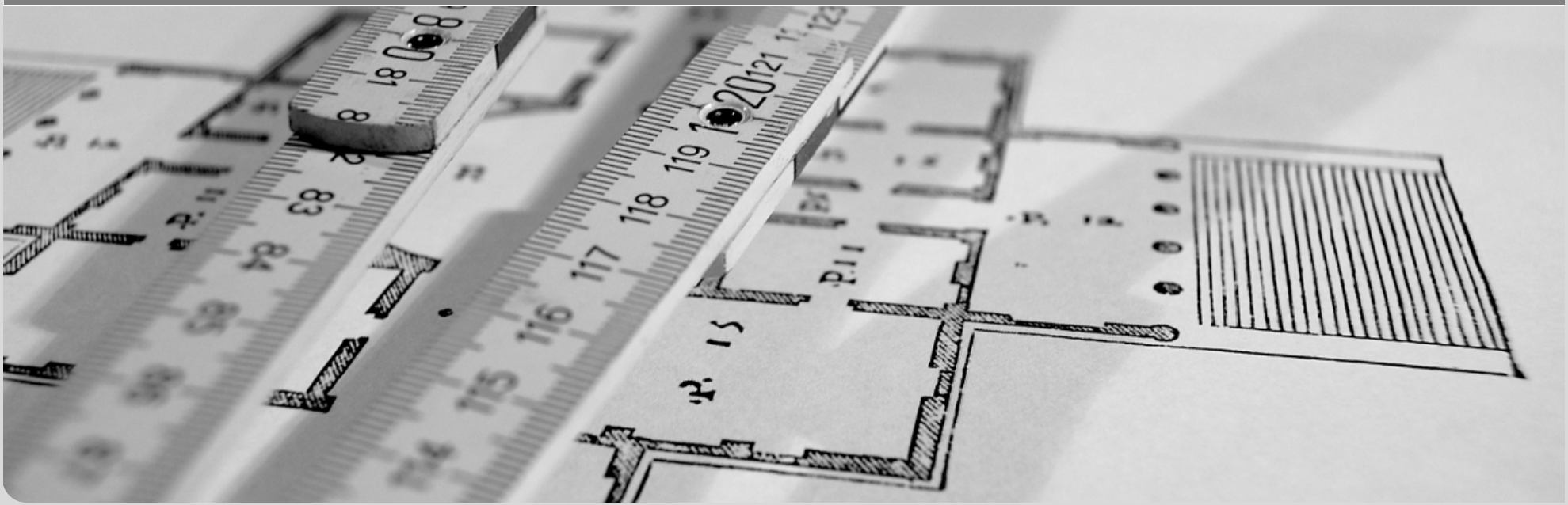
Karlsruher Institut für Technologie

Software Design and Quality

Programmieren
08. Tutorium

Robin Rüde | 12. Januar 2015



Gliederung

1

Blatt 4

4. Blatt

- keine Nutzung von System.out
- Fehler mit `"Error,"` ausgeben und **NICHT** anders
- öffentliche Tests sollten grün sein und nicht gelb (= Test nicht bestanden, wird aber trotzdem angenommen)

4. Blatt - Teil A

- swap: Index finden und dann Werte oder Zellen tauschen
- genau lesen, was gefordert ist
- achtet auf eine korrekte Implementierung der toString-Methode

4. Blatt - Teil C

- Vererbung nutzen - vor allem bei Klassen, die zum Teil gleiche Eigenschaften haben.
- Java-Api: Stack, LinkedList und PriorityQueue
- Enums ersetzen keine Vererbung (zum Beispiel bei Simple/ComplexJob
- Stabilitätstests, wie zum Beispiel leere Dateien, sollten funktionieren ohne dass das Programm abstürzt

2

Interfaces

Syntax

Deklaration:

```
interface Name { Konstanten, Methodenkoepfe }
```

Implementierung

```
class Name implements InterfaceName { ... }
```

Ähnlich wie abstrakte Klassen. Besonderheiten:

- eine Klasse kann mehrere Interfaces implementieren
- enthalten zwei Interfaces die gleiche Methodensignatur, so wird die Methode nur einmal implementiert
- jede Methode der Interfaces muss implementiert werden

Übersicht

```java
/**
 *          Javadoc
 */
interface Vehicle {

        /**
         * Javadoc..
         */
        int getSpeed();
        void accelerate();
        void setNumberOfWheels(int wheelCount);
}
```

- Definition einer Schnittstelle
- keine Instanzen von Interfaces
- Objekte können mehrere Typen haben (Typ des Objekts und mehrere Interfaces)
- Implementierung der Methoden erst in den Klassen

Abstrakte Klassen vs. Interfaces

Abstrakte Klassen

- können Attribute und Methodenimplementierungen beinhalten
- Vermeidung von Code duplication
- Verwendung: partielle Implementierung für gemeinsame Funktionalität

Interfaces

- definieren Schnittstellen (Menge von Methoden, die eine Klasse zur Verfügung stellen muss)
- Verwendung: Abstraktion von konkreter Implementierung

3

Generics

Generics

- Problem: Liste soll für beliebige Typen funktionieren
- aber Implementierung ist für jeden internen Typen gleich
- Lösung: parametrisierte Klassen (aka Generics)
- Typ des gespeicherten Elements ist Parameter
- **keine primitiven Typen!**
- Vorteile
  - Compiler kann Typprüfung durchführen
  - keine redundanten Implementierungen/Code duplication
  - Programm-Code wird lesbarer

Syntax

Syntax und Verwendung

```
class Name<Typ-Parameter> { ... }
interface Name<Typ-Parameter> { ... }
```

Es sind auch mehrere Typparameter möglich.
Einschränkung der zulässigen Typen:

```
class Name<T extends OtherType> { ... }
```

Danach ist T wie jeder andere Klassenname verwendbar:

```
class List<T> {
    List<T> next;
    T item;
    void setItem(T item) { this.item = item; }
    T getItem() { return this.item; }
}
```

Wrapper-Objekte, Autoboxing

- primitive Typen sind als Parameter unzulässig
- Verwendung von Integer, Short, Boolean (Klassen aus `java.lang`)
- Seit Java 5: automatische Konvertierung von primitiven Typ $\leftrightarrow$ Wrapper-Objekt
- deshalb: als Typparameter Wrapper-Objekte aber ganz normal ints speichern und zuweisen

```
1 // Erzeugen von Wrapper-Objekten
2 Integer intObj = new Integer(5);
3 intObj = Integer.valueOf(17);
4 Double doubleObj = new Double(4.5);
5
6 // Autoboxing und unboxing
7 int i = 42;
8 Integer j = i; // expandiert zu j = Integer.valueOf(i); „Boxing“
9 int k = j + 3; // expandiert zu k = j.intValue() + 3; „Unboxing“
```

Generische Methoden

- auch Methoden können generisch sein

```
<Typ-Parameter> Typ Name(Parameter-Liste) { ... }

// Example
class Util {
        public static <T> T randomSelect(T e1, T e2) {
                return (Math.random() > 0.5 ? e1 : e2);
        }
}
enum Enemy { Boss, Sentinel, Drone }
...
Enemy e = Util.<Enemy>randomSelect(Enemy.Drone, Enemy.Sentinel);

// normally using type inference
Enemy e = Util.randomSelect(Enemy.Drone, Enemy.Sentinel);
```

Comparable<Typ>

- Implementierung des Interfaces Comparable<Typ>
- Rückgabe von negativ, 0 oder positiven Wert wenn kleiner, gleich, größer
- Verwendbar für
  `Collections.sort(List<T extends Comparable<T>>)`

4

Rekursion

Rekursion

- Prinzip: Führe gleiche Berechnungsschritte mit kleineren Proplemen aus
- Realisierung: Methoden rufen sich selbst auf
- (Standard-Implementieurung von Divide-And-Conquer)
- genau überlegen, ob die Abbruchbedingung korrekt ist, da sonst endlos Rekursion droht
- StackOverflow wenn Rekursionstiefe zu groß
- Speicherverbrauch linear zur Zahl der Aufrufe

Beispiel

$$n! = n \cdot (n-1) \cdot (\dots) \cdot 1 = n \cdot (n-1)!$$

```
1 public static int fac(int n) {
2         if (n > 0) {
3                 return n * fac(n - 1);
4         } else {
5                 return 1;
6         }
7 }
```

5

Aufgaben

Aufgabe 1: Rekursive Summe

Rekursive Summe

Setze folgende Formel in ein rekursives Programm um:

$f(0) = 0$

$f(n) = f(n-1) + n$

Kann man diese Formel auch ohne Rekursion iterativ oder sogar mit einer expliziten Formel berechnen?

Implementiere falls möglich weitere Methoden, die die obige Funktion iterativ und explizit berechnen.

Aufgabe 2: Generics

Schreibe eine generische Methode (oder eine Methode in einer generischen Klasse), die aus einer java.util.ArrayList eine java.util.LinkedList macht.

Aufgabe 2: Generics

```java
public <T> LinkedList<T> makeLinked(ArrayList<T> in) {
    LinkedList<T> out = new LinkedList<T>();
    for(T e : in) out.add(e);
    return out;
}
```

Ende

Fragen?

Ende

Fragen?
Vielen Dank für eure Aufmerksamkeit!