

## Fragen, Antworten, Kommentare zur aktuellen Vorlesung

Die Online-Befragung zur genutzten alternativen Veranstaltungsform und zur Lehrevaluation ist online. Bitte ausfüllen: <https://forms.gle/fZTBWPmUhK4oqLHw5>. Sie werden eventuell aufgefordert sich bei Google anzumelden, das ist nur notwendig, wenn Sie in der Bearbeitung eine Pause machen und das Teilergebnis zwischenspeichern wollen. Die Befragung endet am 19.12., die Ergebnisse stehen in einem nachfolgenden Fragen&Antworten-Dokument auf der Webseite der Veranstaltung.

Frage: Sollten die Tests zum Stapel eigentlich alle in einer Klasse stehen?

Antwort: Generell werden die Tests einer Klasse immer in einer Testklasse zusammengefasst. Allerdings gibt es keine Regel, dass es so sein muss. Eine Aufteilung ist insbesondere dann sinnvoll, wenn es sich um komplexe zu testende Methoden handelt oder um Methoden, die häufiger in einem Projekt angepasst werden müssen. Dies ist wieder einer der Fragen mir der Antwort, es hängt vom Projektkontext ab. Für das erste Semester gibt es keine Vorgaben. Keine Vorgabe, aber ein Wunsch ist es, dass Testmethoden sprechende Namen enthalten, also im Methodennamen steht was mit wem gemacht wird und welches Ergebnis erwartet wird, z. B. „testPushAufLeerenStapelErlaubt“. (Ich habe dies bei den Messreihentests leider nicht eingehalten.)

Hinweis: Bei Programmen im Praktikum ist mir aufgefallen, dass relativ selten „this.“ genutzt wird. Da wir jetzt alle Schreibweisen kennen, zeigt das folgende Beispiel, wie hilfreich solche Bezeichnungen vor Variablen und Methoden sind. Da dies in der Praxis in ordentlichen Unternehmen immer gefordert wird, sollte dieser Stil frühzeitig genutzt werden. Der Hintergrund ist, dass immer alle entwickelnden Personen eines Teams den Code anderer lesen und bearbeiten können sollen. Andere Programmiersprachen haben vergleichbare Regeln.

```
public void sprechenderMethodenname(String produktname, int id) {
    this.name = produktname; // klar erkennbar, auf der linken Seite ist
                             // eine Objektvariable
    super.id = id;           // klar erkennbar, auf der linken Seite ist
                             // eine Objektvariable die (irgendwoher)
                             // geerbt wurde
    Status.count++;         // klar erkennbar, dass eine Klassenvariable
                             // genutzt wird
    this.weiterleiten(id);  // klar erkennbar, es wird erwartet, dass sich
                             // die Objektmethode in dieser Klasse befindet
    super.markieren(this.name); // klar erkennbar, dass eine Objektmethode
                             // (irgendeiner) beerbten Klasse aufgerufen
                             // werden soll
    Status.sichern(this.name); // klar erkennbar, dass eine Klassenmethode
                             // aufgerufen wird
}
```

Kurz ergänzend zum Thema dynamische Polymorphie, dem zentralen Thema der Objektorientierung, was genauer auch in „nur“ objektbasierten Sprachen wie Go die zentrale Rolle spielt. Durch dynamische Polymorphie können Systeme flexibel gestaltet und einfach zur Laufzeit verändert werden, da jeweils auf die passenden Methoden zugegriffen wird. Da es bei der Praktikumsaufgabe 33 damit vereinzelt Probleme gab, hier nochmals eine andere Beispielskizze. Die Abbildung rechts zeigt eine Vererbungshierarchie, also X3 erbt von X2, weiterhin sind nur die Methodennamen von Methoden angegeben, die in diesen Klassen ausprogrammiert sind. Da alle Klassen von der Klasse Object erben, wurde sie nach oben gesetzt. Generell kann man Variablen eines Typen Objekte dieses Typen oder einer erbenden Klasse zuweisen, also

```
X1 x1 = new X1();
X1 x2 = new X2();
X1 x3 = new X3();
ist alles erlaubt, hingegen
```

```
X3 x31 = new X1();
nicht. Der angegebene Typ auf der linken Seite ist relevant,
wenn es darum geht, welche Methoden aufgerufen werden
können. Für x2 ist z. B. der Aufruf x2.m2() so nicht möglich.
```

Die dynamische Polymorphie kommt ins Spiel, wenn für ein Objekt bestimmt werden soll, welche Methode auszuführen ist. Hierbei wird geschaut, welcher Typ, also welche Klasse zur Erzeugung des Objekts genutzt wurde. Wird `x1.m1()` ausgeführt, wird der mit `//1` markierte Code genutzt, bei `x3.m1()` der mit `//2` markierte Code. Sollte dann eine Methode nicht in der erzeugenden Klasse enthalten sein, wird in der Klasse gesucht, von der die erzeugende Klasse geerbt hat. Wird also `x3.m3()` aufgerufen, wird der mit `//3` markierte Code in der Klasse X2 ausgeführt. Dies ist auch der Grund warum es für jedes Objekt einer neuen Klasse gilt, dass immer die `toString()`-Methode ausgeführt werden kann, da sie in der Klasse Objekt definiert wird.

Wichtig ist, dass dieser Ansatz in der zu Erzeugung genutzten Klasse zuerst nach der Methode zu suchen bei jedem neuen einfachen Methodenaufruf angewandt wird. Beim Aufruf von `x3.m3()` wird die Methode `m3()` in X2 genutzt, beim folgenden Aufruf von `this.m4()` wird wieder in X3 nach der Methode gesucht und an der mit `//4` markierten Stelle gefunden.

Vom erwähnten einfachen Methodenaufruf wird nur abgewichen, wenn vor dem Aufruf `super` steht. Damit wird in der Programmierung festgelegt, dass ausgehend von der aktuell genutzten Klasse K die Methode in der Klasse gesucht wird, von der K geerbt hat. Wird z. B. `x3.m4()` aufgerufen, wird die mit `//5` markierte Methode `m4` in X2 aufgerufen, die dann die mit `//6` markierte Methode `mx` in der Klasse X1 aufruft. Da in dem folgenden Aufruf kein `super` steht, wird für `m1()` wieder zuerst in der Klasse X3 gesucht.

Der Ablauf wirkt auf Personen am Anfang etwas verwirrend, später wird so eine wilde Jagd durch die Klassen selten stattfinden. Oftmals ist die obere Klasse ein Interface, zu dem es dann mehrere Klassen gibt, die dieses Interface implementieren/realisieren. Die Suche nach der auszuführenden Methode ist dann trivial und hängt nur vom zur Erzeugung des Objekts genutzten Klasse ab.

```
public class Object {
    // in Java gegeben
}
```

```
public class X1 {
    public void m1() {} //1
    public void mx() { // 6
        this.m1();
    }
}
```

```
public class X2 extends X1 {
    public void m2() {}
    public void m3() { //3
        this.m4();
    }
    public void m4() { //5
        super.mx();
    }
}
```

```
public class X3 extends X2 {
    public void m1() {} //2
    public void m4() { //4
        super.m4();
    }
    public void m3() {}
}
```

Ein Dialog, den Sie hoffentlich schon mit Ihrer KI geführt haben:

Ich habe Ärger für mehrere Java-Programme bekommen, da wir bei einem if und einem Befehl das in eine Zeile ohne geschweifte Klammern schreibe. Das soll schlechter wartbar sein und die Markierungen von Testüberdeckungen erschweren.

## Erklärung zu deinem Problem mit If-Anweisungen ohne geschweifte Klammern

Du hast recht, dass Java es syntaktisch erlaubt, bei If-Anweisungen (oder auch While/For) die geschweiften Klammern { } wegzulassen, wenn nur ein einzelner Befehl folgt. Zum Beispiel ist `if (bedingung) befehl;` gültig und kompiliert ohne Probleme. Allerdings ist das eine weit verbreitete "bad practice" in der Softwareentwicklung, und deine Kritikpunkte (schlechtere Wartbarkeit und Erschwerung von Testüberdeckungs-Markierungen) sind absolut berechtigt. Ich habe das recherchiert und fasse die wichtigsten Gründe zusammen, basierend auf etablierten Best Practices aus der Community und Coding-Standards.

### Warum ist es schlecht für die Wartbarkeit?

- **Risiko von Fehlern beim Ändern des Codes:** Ohne Klammern hängt der Scope (welche Anweisungen zum If gehören) von der Einrückung ab. Wenn du später einen zweiten Befehl hinzufügst, ohne die Klammern nachzuholen, wird der zweite Befehl *nicht* mehr konditional ausgeführt – was zu subtilen Bugs führt. Ein klassisches Beispiel ist der "Apple SSL Bug" aus 2014, wo ein fehlendes Klammern-Paar zu einer schweren Sicherheitslücke führte. Es verhindert "unforeseen problems when you change the code" und macht den Code "far better to read". Ähnlich warnen Coding-Standards wie MISRA-C:2012 (Regel 15.6), dass das Weglassen von Klammern error-prone ist und vermieden werden sollte. [reddit.com/software/codidact.com](https://reddit.com/software/codidact.com)
- **Lesbarkeit und Konsistenz:** Einrückung allein ist nicht zuverlässig – sie kann in verschiedenen Editoren oder bei Copy-Paste inkonsistent sein. Das "derails the reader's train of thought" und macht es schwerer, sofort zu erkennen, was zum If-Block gehört. Viele Entwickler (z.B. in Reddit-Diskussionen oder Stack Overflow) empfehlen, immer Klammern zu verwenden, um den Code flexibler und verständlicher zu machen. Es ist auch in Lehrmaterialien wie CS Java empfohlen: "It is recommended to always put in the curly braces even if you have just one statement under the if statement." [stackoverflow.com](https://stackoverflow.com)
- **Allgemeine Coding-Standards:** In Java (und ähnlichen Sprachen) raten Style Guides wie Google Java Style oder Oracle's Conventions, Klammern immer zu verwenden. Es reduziert kognitive Last und macht den Code "more maintainable and understandable". [edureka.co/quora.com](https://edureka.co/quora.com)

### Auswirkungen auf Testüberdeckung (Code Coverage)

- **Kein direkter Einfluss auf die Ausführung:** Java behandelt einen Single-Statement-If ohne Klammern identisch zu einem mit Klammern – es gibt

"absolutely no difference" in der Kompilierung oder Laufzeit. Tools wie JaCoCo oder EclEmma (für Code Coverage in Java) messen Branches (If-Zweige) und Line Coverage unabhängig davon, ob Klammern da sind. [stackoverflow.com](https://stackoverflow.com)

- **Indirekte Erschwerung:** Ohne Klammern ist der Code weniger klar strukturiert, was die Interpretation von Coverage-Reports kompliziert macht. In IDEs wie IntelliJ oder Eclipse werden Branches oft visuell markiert (z.B. grün/rot für abgedeckte/nicht abgedeckte Zeilen). Wenn der Scope ambig ist, kann es schwieriger sein, zu verstehen, welche Teile getestet wurden – besonders in komplexen Methoden. Zudem können Tools wie SonarQube oder Checkstyle das Weglassen von Klammern als "Code Smell" flaggen, was die Gesamtcoverage-Bewertung beeinflusst (z.B. durch Maintainability-Metriken). In Team-Umgebungen erschwert es auch, Coverage-Ziele zu erreichen, da unklare Strukturen zu mehr Fehlern in Tests führen. Kurzum: Es macht die Markierungen nicht unmöglich, aber "harmt readability" und kann zu Issues bei Modifikationen führen, was indirekt Tests kompliziert. [softwareengineering.stackexchange.com](https://softwareengineering.stackexchange.com)

**Empfehlung:** Verwende *immer* Klammern für If/Else/Loops, auch bei Single-Statements. Das ist Standard in professionellem Code (z.B. in Open-Source-Projekten oder Unternehmen wie Google). Tools wie Checkstyle können das sogar erzwingen. In deiner Uni- oder Kursumgebung scheint das ein Stil-Vorgabe zu sein – passe deinen Code an, um Punkteabzüge zu vermeiden.