

# PR1, Aufgabenblatt 3

Programmieren I – Wintersemester 2021/22

## Objekte benutzen Objekte über Referenzen; Schleifen

Ausgabedatum: ..... 8. November 2021

### Lernziele

Sicheres Umgehen mit Objektreferenzen; Verstehen von möglichen Fehlersituationen und Ausnahmen; Verstehen und Zeichnen einfacher UML-Diagramme; programmgesteuerte wiederholte Ausführung verstehen; Unterschiede zwischen den verschiedenen Schleifenkonstrukten kennen; Debugger zum Programmverstehen einsetzen können.

### Kernbegriffe

Objekte können Methoden an anderen Objekten aufrufen. Das aufrufende Objekt wird als *Klient* bezeichnet, das aufgerufene als *Dienstleister*. Der Dienstleister definiert in den Signaturen seiner Methoden, welche Parameter für eine Methode akzeptiert werden, d.h. deren Anzahl, Reihenfolge und jeweiligen Typ.

In Java gibt es neben den primitiven Typen *Referenztypen*. Jede in Java geschriebene Klasse definiert einen Referenztyp. Eine Variable mit dem Typ einer Klasse (eine *Referenzvariable*, engl.: reference variable) enthält kein Objekt, sondern lediglich eine *Referenz* (engl.: reference) auf ein Objekt dieser Klasse. Über den Variablenbezeichner können mit der *Punktnotation* die Methoden des referenzierten Objekts aufgerufen werden.

Eine Referenzvariable, die auf kein Objekt verweist, hält den Wert des speziellen Literals `null`. Wird versucht, über eine solche `null`-Referenz eine Methode aufzurufen, dann wird die Programmausführung mit einer so genannten *Ausnahme* (engl.: exception), hier einer `NullPointerException`, abgebrochen. Eine Ausnahme wird immer dann geworfen, wenn während der Ausführung eines Programms ein Fehler festgestellt wird.

In Java können mehrere Methoden einer Klasse den gleichen Namen haben, solange sich ihre Signaturen unterscheiden. Wird auf diese Weise ein Name für mehrere Methoden verwendet, so bezeichnet man den Methodennamen als *überladen* (engl.: overloaded). Gleichnamige Methoden sollten auch eine möglichst ähnliche Semantik haben. Nach den gleichen Regeln kann eine Klasse mehrere Konstruktoren definieren.

Die *Unified Modeling Language (UML)* ist eine grafische Notation zur Beschreibung von Softwaresystemen. Die UML umfasst Diagramme für die Darstellung verschiedener Ansichten auf ein System. Unter anderem gibt es *Klassendiagramme* (engl.: class diagrams) und *Objektdiagramme* (engl. object diagrams). Ein Klassendiagramm beschreibt die statische Struktur eines Systems, indem es die Beziehungen zwischen seinen Klassen darstellt. Ein Objektdiagramm gibt Antwort auf die Frage, welche Struktur ein Teil des Systems zu einem bestimmten Zeitpunkt während der Laufzeit besitzt („Schnappschuss“); es zeigt üblicherweise Objekte mit den Belegungen ihrer Felder (Werte primitiver Typen oder Referenzen).

Neben den Kontrollstrukturen *Sequenz* und *Auswahl* gibt es die *Wiederholung* (engl.: repetition). Diese wird in Java überwiegend durch *Schleifenkonstrukte* (engl.: loop constructs) realisiert, die ermöglichen, dass eine Reihe von Anweisungen mehrfach, nur einmal oder gar nicht ausgeführt wird. Grundlegend lassen sich u.a. *Zählschleifen* (in Java mit `for` möglich) und *bedingte Schleifen* (in Java mit `while` und `do-while` möglich) unterscheiden. Die Wiederholung durch Schleifenkonstrukte wird auch *Iteration* (engl.: iteration) genannt.

Schleifenkonstrukte bestehen immer aus zwei Teilen: dem *Schleifenrumpf* (engl.: loop body), der die zu wiederholenden Anweisungen enthält, und der *Schleifensteuerung* (loop control), die die Anzahl der Wiederholungen bestimmt. Eine Schleife ist *abweisend* oder *kopfgesteuert*, wenn es dazu kommen kann, dass der Schleifenrumpf gar nicht ausgeführt wird; wird der Schleifenrumpf auf jeden Fall mindestens einmal ausgeführt, ist die Schleife *nicht-abweisend* oder *endgesteuert*. Hängt die (jeweils nächste) Ausführung des Schleifenrumpfes von einer Bedingung ab (in Java bei allen Schleifen), kann die Schleife *positiv bedingt* sein („Rumpf ausführen, *solange* die Bedingung zutrifft“) oder *zielorientiert bedingt* („ausführen, *bis* die Bedingung zutrifft“). In Java sind alle Schleifen positiv bedingt.

### Aufgabe 3.1 Überweisungsmanager (Termin 1)

Erinnerst Du Dich an die Klasse `Konto`? Diesmal gibt es wieder eine Kontoklasse. Diese findest Du in der Vorlage *Ueberweisungsmanager* im pub-Bereich.


3.1.1 Sieh Dir die Klasse `Konto` genau an; zuerst die Schnittstelle, anschließend den Quelltext. Wieso hat sie mehr als einen Konstruktor? Wie nennt man das dahinter liegende Konzept? Wie reagiert ein Konto auf eine falsche Benutzung? Was heißt hier überhaupt „falsch“?


3.1.2 Füge dem Projekt nun eine neue Klasse `Ueberweisungsmanager` hinzu. Ein Exemplar dieser Klasse soll einen Betrag von einem Konto auf ein anderes überweisen können. Dazu soll die Klasse eine Methode

```
ueberweisen(Konto quellKonto, Konto zielKonto, int betrag)
```

definieren. Implementiere in ihr das gewünschte Verhalten.

Um bei einem interaktiven Methodenaufruf Objekte zu übergeben, müssen diese zuvor erzeugt worden sein. Per Klick auf die Referenzen in der Objektleiste können diese leicht in den BlueJ-Dialog übertragen werden.

 3.1.3 Teste Deinen Überweisungsmanager interaktiv mit BlueJ. Schau Dir dabei mit dem Objekt-Inspektor die internen Zustände eurer Konten an. Was passiert, wenn Du anstelle eines Kontos einfach `null` eingibst? Wie kannst Du diesen Fehler verhindern? **Halte Deine Erkenntnisse schriftlich fest.**

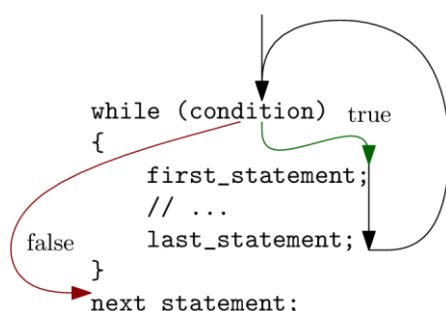
 3.1.4 Was ist die Punktnotation, wofür wird sie verwendet, wie ist sie aufgebaut? **Schriftlich**, am besten anhand eines Beispiels.

### Aufgabe 3.2 Schleifen lesen und verstehen (Termin 1)

3.2.1 Öffne das Projekt *Iteration* und schau Dir die Klasse `Schleifendreher` an. Dort findest Du Beispiele für die verschiedenen Schleifentypen in Java. Führe die Beispiele aus, um Dich mit den Schleifen vertraut zu machen. Beachte dabei die Ausgaben auf der Konsole.

Lies Dir die Methodenrumpfe gründlich durch. Führe sie anschließend im Debug-Modus von BlueJ aus: Öffne dazu die Klasse im BlueJ-Editor und klicke links in die weiße Leiste neben eine Anweisung in der Methode; es erscheint ein kleines rotes Stoppschild. Dies ist ein Haltepunkt. Wenn Du nun wie gewohnt die Methoden aufrufst, öffnet sich der Debugger automatisch und Du kannst den Programmablauf Schritt für Schritt steuern, indem Du auf den Knopf „Schritt über“ (engl. step) klickst. Erkläre Deinen Betreuern, warum die Beispiele zu diesen Ausgaben führen. Konsultiere im Zweifel den zweiten Teil des Skripts.

 3.2.2 Das folgende Diagramm zeigt, wie der Kontrollfluss durch eine `while`-Schleife wandert:



**Zeichne** entsprechende Diagramme für die `do-while`- und die `for`-Schleife. **Schriftlich.**

### Aufgabe 3.3 Eigene Schleifen schreiben (Termin 1)

3.3.1 Schau Dir die Klasse `TextAnalyse` des Projektes *Iteration* an. Dort gibt es eine vorgegebene Methode `istFrage(String text)`, die demonstriert, wie man die Länge eines Strings erhält und wie man auf einzelne Zeichen eines Strings zugreift. Probier diese Methode interaktiv aus, indem Du ein Exemplar von `TextAnalyse` erstellst und dann `istFrage` z.B. mit dem aktuellen Parameter "Wie geht's?" aufrufst.

Vergleiche die Methoden `istFrage` und `istFrageKompakt`. Worin unterscheiden sie sich?

Was passiert, wenn Du der Methode `istFrage` den leeren String als aktuellen Parameter übergibst, also `istFrage("")`? Implementiere eine Lösung, die dies sinnvoll behandelt.

- 3.3.2 Schreibe in der Klasse `TextAnalyse` eine neue Methode `int zaehleVokale(String text)`, die für einen gegebenen Text als Ergebnis liefern soll, wie viele Vokale er enthält. Für den String "hallo" soll die Methode beispielsweise eine 2 zurückgeben. Verwende in der Implementierung einen Schleifenzähler, der bei 0 beginnt und alle Positionen des Strings durchläuft. Die eigentliche Prüfung auf einen Vokal lässt sich am elegantesten mit der switch-Kontrollstruktur lösen – schau Dir im Zweifel das Skript zu Level 2 an, wenn Du nicht genau weißt, wie das switch funktioniert.
- 3.3.3 Schreibe eine weitere Methode `boolean istPalindrom(String text)`, die nur für Palindrome wie *anna*, *otto*, *regallager* oder *axa* *true* liefert. Vergleiche dazu die passenden Zeichen innerhalb des Strings.
- Verwende die Methode `toLowerCase()` aus der Klasse `String`, um den Unterschied zwischen Groß- und Kleinschreibung zu ignorieren, damit auch *Anna*, *Otto* und *Regallager* als Palindrome erkannt werden.
- 3.3.4 **Zusatzaufgabe:** Schreibe eine weitere Methode `String laengstesPalindrom(String text)`, die in der gegebenen Zeichenkette das längste Palindrom ermittelt. Wenn mehrere enthaltene Palindrome am längsten sind, kann ein beliebiges von diesen geliefert werden.

### Aufgabe 3.4 Autorennen (Termin 2)

Kennst Du *Anki Overdrive*? In dieser Aufgabe sollen Autorennen auf einer solchen Spielzeugrennbahn simuliert werden. Unsere Rennbahnen haben vier Spuren, so dass maximal vier Autos an einem Rennen teilnehmen können.

- 3.4.1 Implementiere zuerst eine Klasse `Rennauto`, deren Exemplare einzelne Rennautos modellieren sollen. Ein `Rennauto` soll den *Namen seiner Fahrerin*, seinen *Fahrzeugtyp*, seine *Maximalgeschwindigkeit* und die von ihm *bisher gefahrene Strecke* kennen. Alle Eigenschaften sollen in einem geeigneten Konstruktor initialisiert werden. Außerdem soll ein `Rennauto` die Methode `fahre` anbieten, in der simuliert werden soll, dass das `Rennauto` für einen festen Zeitabschnitt fährt. Innerhalb dieses Zeitabschnitts kann es sich höchstens mit seiner Maximalgeschwindigkeit fortbewegen; es soll aber bei jedem Aufruf der Methode nur mit einer Geschwindigkeit zwischen Null und der Maximalgeschwindigkeit gefahren werden. Dazu kann die Methode `Math.random` benutzt werden, die eine Zufallszahl aus dem halboffenen Intervall  $[0,1)$  liefert. Erzeugt interaktiv einige Exemplare der Klasse und testet mit Hilfe des Objektinspektors, ob alles funktioniert.

Da das wiederholte Übergeben mehrerer Parameter ermüdend ist: definiere einen weiteren Konstruktor, bei dem lediglich der Name der Fahrerin übergeben werden muss; die weiteren Eigenschaften sollen bei diesem Konstruktor Standardwerte erhalten (entscheide selbst, welche).

- 3.4.2 Implementiere anschließend eine Klasse `Rennbahn`, deren Exemplare Rennen mit `Rennautos` durchführen können. Ein neues `Rennbahn`-Exemplar soll als Konstruktorparameter seine *Streckenlänge* erhalten. Mit der Methode `setzeAufSpur` kann ein `Rennauto` übergeben werden, das an einem Rennen teilnehmen soll (und ähnlich wie bei einer Carrera-Bahn eine eigene Spur bekommt). Es können maximal vier `Rennautos` eine eigene Spur bekommen, weitere `Rennautos` sollen ignoriert werden. In der Methode `simuliereZeitabschnitt` sollen alle beteiligten `Rennautos` für einen Zeitabschnitt fahren. Weiterhin sollen implementiert werden: Eine Methode `liefereSieger`, die eine Referenz auf ein `Rennauto` liefert, das bereits die gesamte Streckenlänge gefahren ist; wenn noch keines der Autos im Ziel ist, liefert sie `null`. Eine Methode `rennenDurchfuehren`, die `simuliereZeitabschnitt` so lange aufruft, bis mindestens eines der beteiligten `Rennautos` gewonnen hat.
- 3.4.3 Rennfahrer können auch disqualifiziert werden: Implementiere in der Klasse `Rennbahn` eine Methode `entferne`, die ein als Parameter übergebenes `Rennauto` wieder von der Rennbahn entfernt.



- 3.4.3 **Erstelle ein Objektdiagramm** einer Rennbahn mit drei Teilnehmern (auf Papier). Ein Objektdiagramm stellt einen Ausschnitt aus dem Objektgeflecht zur Laufzeit dar; überlege dafür, welche Objekte in welcher Form zu welchem Zeitpunkt dargestellt werden sollten, um deutlich zu machen, in welchem Zustand das Rennen gerade ist.

### Aufgabe 3.5 Turtle Graphics (Termin 2)

Entpacke das Projekt `TurtleGraphics` in Dein Arbeitsverzeichnis und öffne es in BlueJ. Das Projekt enthält zwei Klassen, `Turtle` und `Dompteur`. Die Klasse `Turtle` stellt Methoden zur Verfügung, mit denen sehr einfach eine Turtle „bewegt“ werden kann; diese Bewegungen werden auf einer Zeichenfläche aufgezeichnet. Die Klasse `Dompteur` enthält eine Methode `start`, in der beispielhaft die Verwendung einer `Turtle` dargestellt ist. Mit Hilfe des Beispiels und der Doku der `Turtle`-Schnittstelle sollen die folgenden Aufgaben gelöst werden.

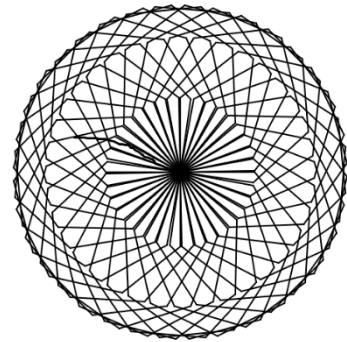
3.5.1 Implementiere eine Methode `zeichneNEck` in `Dompteur`, die mit Hilfe von `Turtle` ein n-Eck zeichnet. Die Anzahl der Ecken und die Kantenlänge sollen als Parameter übergeben werden.

3.5.2 Überlade nun die Methode so, dass es einerseits möglich wird, auch die Position und Farbe des n-Ecks festzulegen, andererseits nur die Anzahl der Ecken festzulegen. Da es schnell lästig wird, eine Methode mit mehreren Parametern interaktiv aufzurufen, solltest Du dafür sorgen, dass sich diese Methoden geeignet gegenseitig aufrufen.

3.5.3 Schreibe eine Methode `zeichneRosette` in `Dompteur`, die mehrere n-Ecke zeichnet, die zueinander jeweils um einen Winkel gedreht sind. Über Parameter soll festgelegt werden können,

- wie viele Ecken die n-Ecke haben sollen,
- wie groß die Kantenlänge der n-Ecke ist,
- um welchen Winkel die n-Ecke zueinander gedreht werden sollen.

Auch hier soll es durch Überladen wieder mehrere Versionen der Methode mit unterschiedlichen Parametern geben. Eine parameterlose soll die eurer Meinung nach schönste Rosette zeichnen.



3.5.4 *Zusatzaufgabe für Kreative:* Schreibe eine weitere Methode in `Dompteur`, die etwas Lustiges, Beeindruckendes, Schönes zeichnet. Sie sollte die Verwendung von Schleifen erfordern.