PR1, Aufgabenblatt 2

Programmieren I – Wintersemester 2021/22

Entwurf eigener Klassen, Fallunterscheidung, Primitive Datentypen, Ausdrücke, Operatoren

Ausgabedatum: ..... 25. Oktober 2021

Lernziele

Neue Klassen definieren können; Exemplare im Konstruktor initialisieren können; Objektzustände mit Feldern modellieren können; Zustände über Methoden verändern und sondieren können; Fallunterscheidungen programmieren können; mit primitiven Datentypen sicher umgehen können; Typumwandlungen erkennen und sie bewerten können; Operatoren anwenden können; logische Ausdrücke verstehen und in Java-Quelltext überführen können.

Kernbegriffe

Der Zustand eines Exemplars ist durch seine *Zustandsfelder* (kurz: *Felder*, engl.: fields) definiert. Mit Hilfe einer speziellen Operation – die *Konstruktor* (engl.: constructor) genannt wird – werden die Felder eines frisch erzeugten Exemplars in einen definierten Anfangszustand gebracht.

Die Methoden einer Klasse definieren das *Verhalten* ihrer Exemplare. Methoden werden unterschieden in *verändernde* und *sondierende Methoden* (engl.: mutator and accessor methods). Verändernde Methoden ändern den Zustand eines Exemplars, sondierende Methoden fragen den Zustand lediglich ab, ohne ihn zu verändern.

Damit sich ein Java-Quelltext übersetzen lässt, muss er der *Java-Syntax* entsprechen. Zusätzlich muss er weitere Regeln befolgen, die nicht in der Syntax ausgedrückt werden. Zum Beispiel müssen Variablen zuerst deklariert werden, bevor man sie benutzen kann. Und an einem Objekt können nur solche Methoden aufgerufen werden, die auch zur Schnittstelle des Objekts gehören.

In Java gibt es den Basisdatentyp `boolean`, der nur *wahr* und *falsch* als *boolesche Werte* definiert (darstellbar u.a. mit den Literalen `true` und `false`), sowie *logische Operationen* auf diesen Werten. Ausdrücke, die als Ergebnis einen booleschen Wert liefern, heißen *boolesche Ausdrücke*.

Sprachmechanismen, die die Reihenfolge der Ausführung von Anweisungen eines Programms steuern, heißen *Kontrollstrukturen* (engl.: control structures). Neben der trivialen Kontrollstruktur *Sequenz* (Anweisungen werden zur Laufzeit in der Reihenfolge ausgeführt, in der sie im Quelltext stehen) gibt es die *Fallunterscheidung*. Bei der Fallunterscheidung mit einer *if-Anweisung* wird abhängig von einer *Bedingung* (dem Ergebnis eines booleschen Ausdrucks) eine Anweisung ausgeführt oder nicht.

Ein *Block* (engl.: block) fasst eine Sequenz von Anweisungen mit geschweiften Klammern zusammen. Ein Block kann überall dort verwendet werden, wo auch eine einzelne Anweisung stehen kann. In einem Block können lokale Variablen deklariert werden.

Der *Typ* einer Variablen legt fest, welche Werte die Variable annehmen kann. Der Typ eines Ausdrucks legt fest, welche Werte die Auswertung des Ausdrucks zur Laufzeit liefern kann. In Java gibt es zwei grundsätzlich unterschiedliche Arten von Typen: eine fest definierte Menge von *primitiven Datentypen* (engl.: primitive types) und *Referenztypen* (engl.: reference types; werden erst auf dem nächsten Aufgabenblatt diskutiert).

Der meistverwendete primitive Datentyp in Java ist `int`; er kann *Ganzzahlen* (engl.: integer numbers) von -2^{31} bis $2^{31}-1$ darstellen, also von -2.147.483.648 bis +2.147.483.647. Obwohl dieser Wertebereich für viele Aufgaben ausreicht, muss immer darauf geachtet werden, ob das auf das eigene Programm zutrifft.

Viele Programmiersprachen bieten Unterstützung für *Gleitkommazahlen* (engl.: floating point numbers). Der Begriff kommt daher, dass die Zahl durch das Gleiten (Verschieben) des Dezimalpunkts als Produkt aus einer Zahl und der Potenz einer Basis dargestellt wird, z.B. $9{,}625_{10} = 1001{,}101_2 = 1{,}001101_2 * 2^3$. Java bietet zwei primitive Typen für Gleitkommazahlen an: `float` (32 Bit), und `double` (64 Bit). Gleitkommazahlen können ebenso überlaufen wie Ganzzahlen; außerdem können sie auch innerhalb des Wertebereichs diejenigen Werte nicht exakt darstellen, die keine Summen von Zweierpotenzen sind (z.B. nicht den dezimalen Wert 0,1). Beim Umgang mit Gleitkommazahlen muss daher besonders auf Wertebereich und Genauigkeitsgrenzen geachtet werden.

Operatoren verknüpfen *Operanden* zu *Ausdrücken* (engl.: expression). Die üblichen Operatoren für Addition, Subtraktion, Multiplikation und Division für numerische Typen sind *binäre* (zweistellige) Operatoren.

In den meisten Programmiersprachen gibt es eine große Zahl von Operatoren, die alle ihre eigenen Vorrangregeln besitzen. Der Multiplikationsoperator in Java besitzt zum Beispiel eine höhere *Präzedenz* (auch Rang) als der Plus-Operator. Die Vorrangregeln bestimmen also die Auswertungsreihenfolge. Der Klarheit halber empfiehlt es sich, in komplizierten Ausdrücken durch Klammern die Auswertungsreihenfolge ausdrücklich anzugeben, auch wenn die Präzedenz schon dieselbe Wirkung hätte (*defensives Klammern*).

Bei einer Zuweisung wird der Wert des Ausdrucks auf der rechten Seite in der Variablen auf der linken Seite abgelegt; dabei müssen der Typ der Variablen und der Typ des Ausdrucks zueinander *kompatibel* sein. Bei den numerischen Datentypen in Java kann es dabei zu impliziten und expliziten *Typumwandlungen* (auch Typkonversion oder -konvertierung, engl.: *type cast* oder *coercion*) kommen. Hat der Zieltyp eine höhere Genauigkeit als der Typ des Ausdrucks, wird die Umwandlung automatisch (implizit) durchgeführt, da keine Informationen verloren gehen können (engl. auch *widening conversion*). Ist der Zieltyp hingegen „enger" als der Ausdruck, muss eine *explizite* Umwandlung durch den Programmierer erzwungen werden, weil Genauigkeit verloren gehen kann (engl. auch *narrowing conversion*). Der gewünschte Typ für eine Typumwandlung wird in runden Klammern vor den umzuwandelnden Ausdruck geschrieben, z.B. `int p = (int) 3.1416`.

Aufgabe 2.1 Der Klassiker: das Konto (Termin 1)

2.1.1 Erzeuge ein neues BlueJ-Projekt mit dem Titel *Bank*. Erzeuge darin eine Klasse `Konto`, die Bankkonten modellieren soll.

2.1.2 Der Zustand eines Kontos soll durch einen *Saldo* definiert sein, der den aktuellen Kontostand angibt. Definiere ein Feld vom Typ `int` für den Saldo und sorge für eine korrekte Initialisierung.

2.1.3 Definiere zwei Methoden zum Einzahlen und Abheben, `void einzahlen(int)` und `void abheben(int)`, die den Kontostand jeweils neu berechnen und den Zustand entsprechend verändern. Überprüfe, ob deine Methoden funktionieren, indem du in BlueJ den Zustand der Exemplare vor und nach der Ausführung der Methoden betrachtest.

2.1.4 Der Kontostand soll durch Klienten abgefragt werden können. Implementiere eine Methode, die die Belegung des Feldes zurückliefert. Welche Methoden der Klasse `Konto` sind nun sondierende Methoden, welche verändernde?

2.1.5 Standardmäßig darf ein Konto nicht überzogen werden. Wie kann im Quelltext das Überziehen verhindert werden? Implementiere diese Absicherung.

2.1.6 Die Bank will neue Kunden werben und spendiert jedem Neukunden 10 Euro. Implementiere einen Konstruktor, der den Saldo eines neu erzeugten Kontos auf diesen Betrag setzt.

2.1.7 Jedes Konto soll eine Kontonummer erhalten, die ein Klient nur einmalig beim Anlegen des Kontos setzen kann. Implementiere dies mit Hilfe einer Exemplarkonstanten. Warum muss es für die Kontonummer auch eine sondierende Methode geben?

2.1.8 Damit auf den ersten Blick erkennbar ist, wer die Klasse `Konto` programmiert hat und was modelliert wird, sollst du nun den Kommentar am Beginn des Quelltextes verändern und eine Beschreibung der Klasse einfügen. Wechsel im Editor die Ansicht von „Quelltext" auf „Dokumentation". Wird dein Kommentar sichtbar? Wenn nein, warum nicht? Wenn ja, warum?

Bereite deinen Quelltext für die Abnahme vor. Im pub liegen hierfür die Quelltextkonventionen in einem PDF-Dokument bereit, das du dir durchlesen solltest. Diese Quelltextkonventionen beschreiben sinnvolle Regeln zur Vereinheitlichung und guten Lesbarkeit von Programmtext**. Ab jetzt gilt: Nur Quelltexte, die diesen Regeln entsprechen, werden von Betreuern und Betreuerinnen abgenommen!**

2.1.9 *Zusatzaufgabe*: Implementiere einen *Dispo* für das Konto, also die Möglichkeit für einen Dispositionskredit. Er muss als ganze, positive Zahl bei der Erzeugung angegeben werden, soll aber später auch geändert werden können. Jetzt kann das Konto bis zu diesem Dispo überzogen werden. Wie gehst Du damit um, wenn der Dispo bei bestehender Überziehung in die falsche Richtung (was heißt falsch hier?) geändert wird? Konstruiere eine Aufrufreihenfolge durch einen Klienten, die in diese Situation führt, und halte sie für die Abnahme schriftlich fest.

Aufgabe 2.2 Ratemaschine (Termin 1)

2.2.1 Schreib eine Klasse `Ratemaschine`. Bei einem Exemplar dieser Klasse soll ein Klient eine ganze Zahl raten können. Die zu ratende Zahl soll beim Erzeugen eines Exemplars übergeben werden.

2.2.2 Implementiere eine Methode `istEsDieseZahl`, der man als Parameter eine Zahl als Rateversuch übergibt. Als **Rückgabewert** liefert die Ratemaschine einen String mit dem Inhalt „Zu niedrig geraten!“ oder „Zu hoch getippt!“ bzw. „Stimmt!“.

Testet vor der Abnahme eure Klassen innerhalb eures Paares gegenseitig, indem eine Person von euch ein Exemplar der Klasse erzeugt, während die andere wegsieht. Die andere Person soll anschließend die Zahl durch fortlaufende Aufrufe der Methode `istEsDieseZahl` erraten. Nicht mit dem Objektinspektor schummeln!

2.2.3 Erweitere die Ratemaschine, so dass sie die Anzahl der Rateversuche festhält, und, sobald man die richtige Zahl getippt hat, nicht nur „Stimmt!“ zurückgibt, sondern zusätzlich, wie viele Versuche man gebraucht hat.

Tipp: Strings lassen sich in Java mit primitiven Datentypen über den + Operator verbinden:
`"Hallo " + 42`

2.2.4 *Zusatzaufgabe*: Die Zahl der Versuche soll durch die Maschine bewertet werden. Mittels einer switch-Anweisung sollen Bewertungen über die Rateleistung zurückgegeben werden. Bei 1 bis 5 Versuchen soll ‚Das war doch nur Glück!’ geliefert werden. Bei 6 bis 10 Versuchen soll ‚Gar nicht mal so gut!’ auftauchen und bei mehr als 10 Versuchen sollte zu einem anderen Hobby geraten werden. Die Syntax von switch könnt ihr hier nachschauen (englische Spezifikation): http://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html#jls-14.11.

Ein sinnvolles Beispiel (ebenfalls auf Englisch) findet ihr auf http://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html.

Aufgabe 2.3 Digitale Waagen (Termin 2)

Heutzutage gibt es digitale Körpergewicht-Waagen zum Schleuderpreis. Um in diesem Markt etwas Besonderes bieten zu können, wollen wir eine Java-Klasse schreiben, deren Exemplare in einer modernen Waage zum Einsatz kommen sollen und den Besitzer über seinen Fortschritt bei der Gewichtskontrolle informieren.

2.3.1 Lege ein neues Projekt *Fitness* an und darin eine neue Klasse `Waage`. Diese soll über einen Konstruktor verfügen, der das aktuelle Körpergewicht der Person als Parameter mit dem Typ `int` entgegennimmt und in einer Exemplarvariablen `_letztesGewicht` hinterlegt. Mach dir Gedanken darüber, in welcher Einheit du das Gewicht speichern willst, beispielsweise Gramm oder Kilogramm. Mach dies für Klienten deutlich, indem du **wie immer entsprechende Schnittstellen-Kommentare** schreibst!

2.3.2 Schreib eine Methode `void registriere(int neuesGewicht)`. Diese wird jedes Mal aufgerufen, wenn der Besitzer sich erneut wiegt. Als Parameter bekommt sie das Ergebnis der physischen Messung der Waage übergeben.

2.3.3 In der Methode `registriere` soll festgestellt werden, ob sich das Gewicht seit der letzten Messung verändert hat. Diesen Trend sollst du im Zustand des Exemplars festhalten. Implementiere anschließend eine parameterlose Methode `gibTrend` mit dem Ergebnistyp `int`, welche folgendes zurückgeben soll:

-1, falls der Besitzer leichter geworden ist

1, falls er schwerer geworden ist

0 sonst

2.3.4 Implementiere zwei parameterlose Methoden `gibMinimalgewicht` und `gibMaximalgewicht`, die als Ergebnis vom Typ `int` die extremen Messwerte der bisherigen Messreihe eines Objekts zurückgeben.

2.3.5 Implementiere eine parameterlose Methode `gibDurchschnittsgewicht`, die das durchschnittliche Gewicht über alle Messungen eines Objekts bildet und diesen als Ergebnistyp `int` zurückgibt. **Beachte dabei**: Zur Berechnung dürfen nur die bisher vorgestellten Konzepte verwendet werden.

Aufgabe 2.4 Boolesche Ausdrücke (Termin 2)

Die folgende Tabelle zeigt alle booleschen Operatoren in Java:

x	y	x `&&` y	x `||` y	x `^` y	x `==` y	x `!=` y	`!`x
false	false	false	false	false	true	false	true
false	true	false	true	true	false	true	true
true	false	false	true	true	false	true	false
true	true	true	true	false	true	false	false

2.4.1 Lade das Projekt *Ampeln* aus dem pub-Bereich herunter und studiere die Klasse `Ampel`. **Zeichne** alle vier in Deutschland üblichen Ampelphasen für den Kraftverkehr in der richtigen Reihenfolge auf.

2.4.2 Im `Ampel`-Konstruktor wird nur eines der drei booleschen Felder initialisiert. Warum ist es technisch nicht notwendig, die anderen beiden Felder zu initialisieren? Füge von Hand explizite Initialisierungen hinzu, um die Lesbarkeit des Quelltexts zu erhöhen.

2.4.3 Analysiere die Rümpfe der drei Methoden `leuchtetRot`, `leuchtetGelb` und `leuchtetGruen` und überzeuge dich von ihrer prinzipiellen Äquivalenz (Gleichwertigkeit). Warum sind weder der Vergleich mit *true* in `leuchtetRot` noch die beiden Fallunterscheidungen in `leuchtetRot` und `leuchtetGelb` notwendig?

2.4.4 Die Methode `schalteWeiter` funktioniert im Auslieferungszustand lediglich für den Wechsel von der ersten in die zweite Phase. Vervollständige den Methodenrumpf für alle vier Phasenwechsel. Teste die Methode, indem du ein Ampel-Exemplar erzeugst, dieses per Doppelklick mit dem Objektinspektor öffnest und anschließend vier Mal weiterschaltest. Optisch ansprechender ist das Testen nach der Erzeugung eines Exemplars von `AmpelGUI`. Hier wird per Knopfdruck auf „weiter“ in die nächste Ampelphase geschaltet.

2.4.5 Die Klasse `Bmpel` besitzt keine booleschen Felder für die Lampen, sondern verwendet stattdessen ein `int`-Feld, welches periodisch die Werte 0, 1, 2, 3 durchläuft. Ein Testen mit dem Objektinspektor ist jetzt nicht mehr sinnvoll, aber dafür gibt es ja die Klasse `BmpelGUI`. Die Implementation der `schalteWeiter`-Methode ist im Vergleich mit den vorherigen Lösungen trivial. Schaue dir ihren Rumpf an und probiere in der BlueJ-Direkteingabe (mit Strg+E aktivierbar) aus, was die Formel `(x + 1) % 4` für verschiedene `x` bewirkt.

Anmerkung: In Java wird die Modulo-Operation durch den Operator `%` realisiert.

2.4.6 Die drei Methoden zur Abfrage der Lampen gestalten sich jetzt etwas umfangreicher, da das Ergebnis nicht bereits in einem Feld vorliegt, sondern erst berechnet werden muss. Die Methode `leuchtetGruen` ist schon fertig implementiert. Vervollständige die Rümpfe der anderen beiden Methoden.

2.4.7 Die Klasse `Zmpel` kommt vollständig ohne Fallunterscheidungen aus. In der Methode `schalteWeiter` wird der Folgezustand direkt aus dem aktuellen Zustand berechnet, indem die Felder mit passenden booleschen Operatoren verknüpft werden. Bei der Suche nach den Formeln ist es hilfreich, sich eine Tabelle anzulegen, die den aktuellen Zustand auf den Folgezustand abbildet:

Gültige Zustände			Folgezustände		
rot	gelb	grün	rot	gelb	grün
false	false	true			
false	true	false			
true	false	false			
true	true	false			

grün im Folgezustand ist offenbar genau dann true, wenn rot und gelb true sind, d.h.
`_gruen = _rot && _gelb;`

Füll die beiden offenen Spalten aus und vervollständige anschließend die Implementation.

Zusatzaufgabe 2.5 Wann ist Ostern?

2.5.1 Öffne das Projekt *Osterauskunft*. Darin findest du eine Klasse `Osterauskunft` mit einer vorgegebenen, leeren Methode `int wannIstOsternImJahr(int jahr)`. Die Methode soll für ein übergebenes Jahr den Tag des Monats liefern, an dem Ostersonntag ist. Da dieser im März oder April liegen kann und wir zwischen den beiden Monaten unterscheiden müssen, sollen **negative Zahlen** für den März und **positive Zahlen** für den April geliefert werden.

Implementiere in dieser Methode die *Gaußsche Osterregel*, die folgendermaßen definiert ist:

Gaußsche Osterregel

Für eine gegebene Jahreszahl *J* berechne man die Zahlen *m* und *n* wie folgt:

- Für 1800 <= *J* <= 1899 ist *m* = 23, *n* = 4.
- Für 1900 <= *J* <= 2099 ist *m* = 24, *n* = 5.
- Für 2100 <= *J* <= 2199 ist *m* = 24, *n* = 6.

Falls *J* nicht innerhalb der angegebenen Grenzen liegt, ist ein Wert zurückzugeben, welcher ein ungültiges Datum repräsentiert, etwa 0 oder 100. Dokumentiert dies im Schnittstellenkommentar.

Andernfalls bezeichne man die Reste der Divisionen

- *J* modulo 19 mit *a*,
- *J* modulo 4 mit *b*,
- *J* modulo 7 mit *c*,
- (19**a* + *m*) modulo 30 mit *d*,
- (2**b* + 4**c* + 6**d* + *n*) modulo 7 mit *e*.

Falls 22 + *d* + *e* kleiner gleich 31 ist, dann fällt der Ostersonntag auf den (22 + *d* + *e*)-ten März, ansonsten auf den (*d* + *e* - 9)-ten April.

Folgende Zusatzregeln sind zu beachten:

- Anstelle des 26. Aprils ist immer der 19. April zu setzen.
- Anstelle des 25. Aprils ist der 18. April zu setzen, falls sowohl *d* = 28 als auch *e* = 6 als auch *a* >= 10 sind.

Im Jahr 1985 fiel der Ostersonntag auf den 7. April, daher sollte deine Implementation hier die Zahl 7 zurückliefern. Im Jahr darauf war es der 30. März, also ist hier -30 das korrekte Ergebnis (man beachte das negative Vorzeichen, welches den März signalisiert!).

Um deine Implementation auf Korrektheit zu überprüfen, erstellst du ein Exemplar der vorgegebenen Klasse `Osterpruefer`. Daraufhin öffnet sich automatisch ein neues Fenster, in dem die fehlerhaft berechneten Daten als anklickbare Knöpfe angezeigt werden. Die Textfarbe innerhalb des Knopfes gibt dabei Aufschluss über die Art des Fehlers. Klicke einen beliebigen Knopf an, um zu erfahren, warum das Datum als fehlerhaft erkannt wurde.