SE1 Aufgabenblatt 5

Softwareentwicklung I – Wintersemester 2015/16

Primitive Datentypen, Ausdrücke, Operatoren

MIN-CommSy-URL: ..... https://www.mincommsy.uni-hamburg.de/

Projektraum: ..... SE1 CommSy WiSe 15/16

Ausgabedatum: ..... 12. November 2015

Kernbegriffe

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).

Der meistverwendete primitive Typ in Java ist `int`; er kann *Ganzzahlen* 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.

Ein anderer primitiver Datentyp sind die *Gleitkommazahlen*. 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 implementiert den IEEE-Standard 754 für Gleitkommazahlen und definiert somit zwei Gleitkommadatentypen: `float` für einfache Genauigkeit (engl. single precision, 32 Bit), und `double` für doppelte Genauigkeit (engl. double precision, 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 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 (fast immer, siehe Gleitkomma-Aufgabe) 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`.

Lernziele

Mit primitiven Datentypen sicher umgehen können; Verständnis für mögliche Genauigkeitsverluste entwickeln; Typumwandlungen erkennen und sie bewerten können; Operatoren anwenden können; Logische Ausdrücke verstehen und in Java-Quelltext überführen können.

Aufgabe 5.1 Digitale Waagen

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.

5.1.1 Legt 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. Macht Euch Gedanken darüber, in welcher Einheit ihr das Gewicht speichern wollt, beispielsweise Gramm oder Kilogramm. Macht dies für Klienten deutlich, indem ihr **wie immer entsprechende Schnittstellen-Kommentare** schreibt!

5.1.2 Schreibt 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.

5.1.3 In der Methode `registriere` soll festgestellt werden, ob sich das Gewicht seit der letzten Messung verändert hat. Diesen Trend sollt ihr im Zustand des Exemplars festhalten. Implementiert 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

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

5.1.5 *Zusatzaufgabe*: Implementiert eine parameterlose Methode `gibDurchschnittsgewicht`, die das durchschnittliche Gewicht über alle Messungen eines Objekts bildet und diesen als Ergebnistyp `int` zurückgibt. **Herausforderung**: Verwendet zur Berechnung nur bisher in SE1 bekannte Konzepte.

Aufgabe 5.2 Boolesche Ausdrücke

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

5.2.1 Ladet das Projekt `Ampeln` aus dem CommSy herunter und studiert die Klasse `Ampel`. Zeichnet alle 4 in Deutschland üblichen Ampelphasen für den Kraftverkehr in der richtigen Reihenfolge auf.

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

5.2.3 Analysiert die Rümpfe der drei Methoden `leuchtetRot`, `leuchtetGelb` und `leuchtetGruen` und überzeugt euch von ihrer softwaretechnischen Äquivalenz. Warum sind weder der Vergleich mit *true* in `leuchtetRot` noch die beiden Fallunterscheidungen in `leuchtetRot` und `leuchtetGelb` notwendig?

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

5.2.5 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.

5.2.6 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. Vervollständigt die Rümpfe der anderen beiden Methoden.

5.2.7 *Zusatzaufgabe*: 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
false	true	false			false
true	false	false			false
true	true	false			true

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

Füllt die beiden offenen Spalten aus und vervollständigt anschließend die Implementation.

Aufgabe 5.3 Vorsicht bei Gleitkommazahlen

Für diese Aufgabe gibt es eine separat ausgedruckte Tabelle. Sollte diese Tabelle in eurem Labor nicht zur Verfügung stehen, könnt ihr die Tabelle auch einfach selber im CommSy herunterladen und ausdrucken.

5.3.1 Schaut euch die Spalte 1 („Ausdruck“) in der Tabelle an und notiert in **Spalte 2** („Gesunder Menschenverstand“), welches Ergebnis ein Mensch erwarten würde. Bitte ignoriert ganz bewusst evtl. vorhandenes Wissen über die interne Darstellung von Gleitkommazahlen oder spezielle Rechenregeln von Java!

Dies erledigt ihr bitte für alle Zeilen in der Tabelle, bevor ihr mit der nächsten Teilaufgabe weitermacht!

5.3.2 Ladet euch das Programm `frepl` aus dem CommSy herunter und startet es per Doppelklick auf die jar-Datei. Falls das auf eurem Betriebssystem nicht funktionieren sollte, startet eine Konsole und gebt ein:

```
java -jar frepl.jar
```

Befüllt jetzt mit Hilfe des Programms `frepl` die **Spalte 3** der Tabelle („Was der Computer liefert“).

5.3.3 Befüllt die **Spalte 4** („Begründung für das abweichende Verhalten“) so gut es euer Kenntnisstand erlaubt. Keine Sorge, falls ihr nicht alles sofort versteht; in der Diskussion mit dem Betreuer sollte es klar werden!

Aufgabe 5.4 Quadratische Gleichungen lösen

5.4.1 Legt ein neues Projekt an und erstellt darin eine neue Klasse `QuadratischeGleichung`. Der Sinn dieser Klasse ist das Lösen quadratischer Gleichungen der Form $ax^2 + bx + c = 0$ (zum Beispiel mit der pq-Formel).

5.4.2 Schreibt einen Konstruktor mit drei formalen Parametern für die Koeffizienten a, b und c der Gleichung.

5.4.3 Schreibt eine parameterlose Methode namens `ersteNullstelle`, welche die erste Nullstelle der quadratischen Gleichung zurückliefert. Beispiel:

```
QuadratischeGleichung qg = new QuadratischeGleichung(4, 5, -6);
qg.ersteNullstelle()    // liefert -2.0 als Ergebnis
```

5.4.4 Schreibt eine parameterlose Methode namens `zweiteNullstelle`, welche die zweite Nullstelle der quadratischen Gleichung zurückliefert. Beispiel:

```
qg.zweiteNullstelle()   // liefert 0.75 als Ergebnis
```

5.4.5 Nicht alle quadratischen Gleichungen haben zwei Nullstellen, d.h. nicht jede Parabel schneidet die X-Achse an genau zwei Stellen! Welche weiteren Fälle gibt es noch?

5.4.6 Schreibt eine Methode `anzahlNullstellen`, welche die Anzahl der Nullstellen zurückliefert.