SE1 Aufgabenblatt 4

Softwareentwicklung I – Wintersemester 2011/2012

Primitive Datentypen, Ausdrücke, Operatoren

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

Projektraum: ..... SE-1 CommSy WiSe 11/12

Ausgabedatum: ..... 10. November 2011

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 n = (int) 3.1415`.

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 4.1 Ratemaschine

4.1.1 Schreibt 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.

4.1.2 Implementiert 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 eure Klasse innerhalb eures Paares, 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!

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

4.1.4 *Zusatzaufgabe*: Die Zahl der Versuche soll durch die Maschine bewertet werden. Mittels einer switch-Anweisung sollen zusätzlich Bewertungen über die Rateleistung zurückgegeben werden. Bei 1 bis 5 Versuchen soll ‚Das war doch nur Glück!' angegeben 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 der switch-Anweisung könnt ihr hier nachschauen:

http://java.sun.com/docs/books/jls/third_edition/html/statements.html#14.11

Ein Beispiel findet sich zu Beginn der Seite

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/switch.html

Aufgabe 4.2 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

4.2.1 Ladet das Projekt `Ampeln` aus dem CommSy herunter und studiert die Klasse `Ampel`. Macht euch die vier Phasen einer Ampel und ihre Übergänge anhand einer Grafik deutlich.

4.2.2 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.

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

4.2.4 Die Methode `schalteWeiter` schafft im Auslieferungszustand lediglich 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.

4.2.5 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 aus, was die Formel `(x + 1) % 4` für verschiedene `x` bewirkt.

4.2.6 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.

4.2.7 *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 4.3 Vorsicht bei Gleitkommazahlen

Öffnet das Projekt *Gleitkommazahlen*. Es enthält eine (absichtlich nicht übersetzte) Klasse `Nanu` mit mehreren Methoden, die Ihr euch *sehr genau* ansehen sollt. **Versucht, den Quelltext zu verstehen und in eurem Kopf „auszuführen", bevor ihr die Klasse übersetzt. Nehmt euch maximal 45 Minuten Zeit für diese Aufgabe.**

4.3.1 Welche Ergebnisse erwartet ihr beim Aufruf der Methoden? **Schreibt diese in den vorbereiteten Schnittstellenkommentar** der Methoden.

4.3.2 Erzeugt ein Exemplar der Klasse `Nanu` und führt die Methoden aus. Wahrscheinlich seid ihr von den Ergebnissen überrascht. Kommentiert nun (analog zum erwarteten Verhalten) das tatsächliche Verhalten aller Methoden.

4.3.3 Bereitet euch darauf vor mit euren Betreuern zu diskutieren, warum das tatsächliche Verhalten vom erwarteten Verhalten abweicht. Der Debugger von BlueJ dürfte dabei eine große Hilfe sein. **Schreibt eure Begründung** wieder in den Schnittstellenkommentar.
Falls ihr nicht versteht, wie das tatsächliche Ergebnis zustande kommt, dann könnt ihr eure Betreuer befragen. Ihr solltet dann aber zukünftig erst recht einen Bogen um die Verwendung von float und double machen.

Aufgabe 4.4 Wann ist Ostern?

4.4.1 Öffnet das Projekt *Osterauskunft*. Darin findet ihr 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.

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

Gaußsche Osterregel

Eine Jahreszahl *J* ist zu vereinbaren. Dazu bestimme man 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*.

Dann fällt der Ostersonntag auf den (22 + *d* + *e*)-ten März oder 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 eure 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 eure Implementation auf Korrektheit zu überprüfen, erstellt ihr 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. Klickt einen beliebigen Knopf an, um zu erfahren, warum das Datum als fehlerhaft erkannt wurde.

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