SE1, Aufgabenblatt 4

Softwareentwicklung I – Wintersemester 2015/16

Entwurf eigener Klassen, Fallunterscheidung

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

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

Ausgabedatum: ..... 5. November 2015

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.

In Java gibt es den Basisdatentyp `boolean`, der nur die *booleschen Werte* `true` (wahr) und `false` (falsch) definiert, 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.

Neben Variablen mit änderbaren Werten gibt es auch *Konstanten*. Einer Konstanten wird genau einmal bei ihrer Initialisierung ein Wert zugewiesen, der danach unveränderlich ist. Um Variablen als Konstanten zu deklarieren, wird in Java das Schlüsselwort `final` verwendet. Felder, die als Konstanten deklariert werden, nennt man *Exemplarkonstanten*. Exemplarkonstanten können im Konstruktor initialisiert werden, danach lassen sie sich nicht mehr ändern.

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.

Lernziele

Neue Klasse definieren können; Exemplare im Konstruktor initialisieren können; Objektzustände mit Feldern modellieren können; Zustände über Methoden verändern und sondieren können; Fallunterscheidungen programmieren können.

Aufgabe 4.1 Der Klassiker: das Konto

4.1.1 Erzeugt ein neues Projekt mit dem Titel *Bank* und speichert es in eurem Verzeichnis. Erzeugt darin eine Klasse `Konto`, die Bankkonten modellieren soll.

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

Bitte benutzt für Geldbeträge niemals float oder double! Warum nicht? Blendet die BlueJ-Direkteingabe über Ansicht/Direkteingabe anzeigen (Strg+E) ein und gebt anschließend 0.1+0.2 ein. Überzeugt? ;)

4.1.3 Definiert zwei Methoden, `void zahleEin(int)` und `void hebeAb(int)`, die den richtigen Kontostand berechnen und den Zustand entsprechend verändern. Überprüft, ob eure Methoden funktionieren, indem ihr in BlueJ den Zustand der Exemplare vor und nach der Ausführung der Methoden betrachtet.

4.1.4 Der Kontostand soll nun von außen abgefragt werden können (z.B. interaktiv in BlueJ). Implementiert eine Methode, die die Belegung des Feldes zurückliefert. Verwendet als Anhaltspunkt die Beispielmethode.

4.1.5 Welche Methoden des Kontos sind sondierende Methoden, welche verändernde?

4.1.6 Damit auf den ersten Blick erkennbar ist, wer die Klasse `Konto` programmiert hat und was modelliert wird, sollt ihr nun den Kommentar am Beginn des Quelltextes verändern und eine Beschreibung der Klasse einfügen. Wechselt im Editor die Ansicht von „Implementierung“ auf „Schnittstelle“. Wird euer Kommentar sichtbar? Wenn nein, warum nicht? Wenn ja, warum?

4.1.7 Die Bank will neue Kunden werben und spendiert jedem Neukunden 10 Euro. Implementiert einen Konstruktor, der den Saldo eines neu erzeugten Kontos auf diesen Betrag setzt. Hinweis: Der vorgegebene Konstruktor muss nur verändert werden.

4.1.8 Jedes Konto soll eine Kontonummer erhalten, die nur beim Anlegen des Kontos gesetzt werden kann. Implementiert dies mit Hilfe einer Exemplarkonstanten. Warum muss es für die Kontonummer auch eine sondierende Methode geben?

4.1.9 Bereitet euren Quelltext für die Abnahme vor. Im CommSy liegen hierfür die Quelltextkonventionen in einem PDF-Dokument bereit, das ihr euch durchlesen solltet. 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!**

Aufgabe 4.2 Ratemaschine

4.2.1 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.

4.2.2 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!

4.2.3 Erweitert 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`

Aufgabe 4.3 Ein einfacher Taschenrechner

4.3.1 Schreibt eine Klasse `Taschenrechner`. Exemplare dieser Klassen sollen sich ein bisher berechnetes Zwischenergebnis merken können. Zu Beginn soll dieses Zwischenergebnis 0.0 (`double`) betragen.

4.3.2 Durch Aufruf einer Methode `addiere` kann der Benutzer eine übergebene Zahl auf das Zwischenergebnis draufaddieren. Wenn der Benutzer beispielsweise zunächst `addiere(5)` und anschließend `addiere(7)` und zuletzt `addiere(9)` aufruft, dann soll das Zwischenergebnis 21.0 betragen.

4.3.3 Schreibt eine Methode `gibZwischenergebnis`, die das Zwischenergebnis zurückliefert.

4.3.4 Schreibt weitere Methoden für die Subtraktion, Multiplikation und Division.

4.3.5 Es ist recht nervig, nach jeder Rechenoperation erneut die Methode `gibZwischenergebnis` aufrufen zu müssen. Ändert die Methoden `addiere`, `subtrahiere` etc. so ab, dass sie das neue Zwischenergebnis selber zurückliefern.

4.3.6 Schreibt eine Methode `loesche`, die den Taschenrechner auf den Anfangszustand zurücksetzt.

4.3.7 Denkt bitte daran, die Klasse und die Methoden mit Schnittstellenkommentaren zu versehen!

Aufgabe 4.4 Ein wissenschaftlicher Taschenrechner

4.4.1 Wissenschaftliche Taschenrechner bieten in der Regel nicht nur die vier Grundrechenarten an, sondern auch viele andere mathematische Funktionen wie Betrag, Wurzel, Potenz, Sinus, Cosinus…

Um den Betrag einer Zahl x zu bestimmen, kann man in Java `Math.abs(x)` schreiben. Diese und andere Funktionen werden auf der Seite http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html beschrieben. Lest euch die Seite durch und baut dann weitere interessante Funktionen in den Taschenrechner ein!